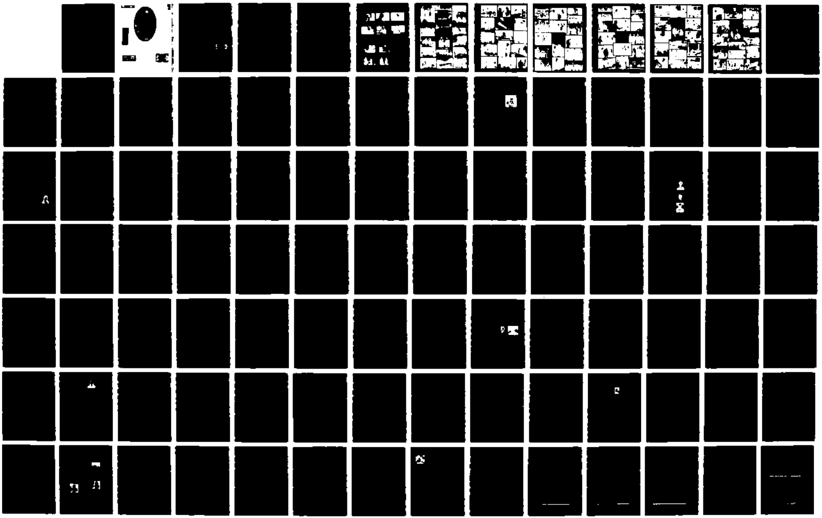
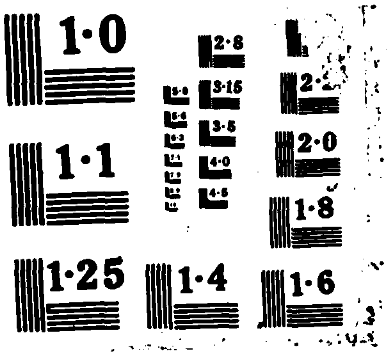
AD-A190 936 PROCEEDINGS OF THE NATIONAL CONFERENCE ON ADA (TRADE MARK) TECHNOLOGY (6TH) HELD ON 14-18 MARCH 1988 IN ARLINGTON VA(U) NORFOLK STATE UNIV VA MAR 88 1/7

UNCLASSIFIED F/G 12/5 NL




1·0
2·8
3·15
3·6
4·0
4·5
2·2
2·0
1·1
1·8
1·25
1·4
1·6

②

PROCEEDINGS OF
SIXTH NATIONAL CONFERENCE ON
ADA TECHNOLOGY

Sponsored by
United States Army
United States Navy
United States Air Force
United States Marine Corps
Ada Joint Program Office

Co-Hosted By
Norfolk State University   University of Maryland

Marriott Crystal Gateway Hotel, Arlington, VA
March 14-17, 1988

Approved for Public Release: Distribution Unlimited

® Ada is a registered trademark of the U.S. Government, Ada Joint Program Office (AJPO)

6th ANNUAL NATIONAL CONFERENCE ON ADA TECHNOLOGY

CONFERENCE COMMITTEE 1987-1988

Elmer F. Godwin, Director
GEF Associates

Phil Andrews
Space & Naval Warfare
Systems Command

MAJ George Bedar
Marine Corps Tactical
Systems
Support Activity

Brian Baker
Navy Department, Pentagon

Miguel A. Carrio, Jr.
Teledyne Brown

Michael Danko
RCA

Dr. Mary Ellis
Hampton University

Donald C. Fuhr
Tuskeegee University

Judith Giles
Intermetrics, Inc.

Dee Graumann
General Dynamics, DSD

Charlene Hayden
GTE Government Systems

Donald E. Hocking
AIRMICS

Dr. Genevieve M. Knight
Coppin State College

MAJ Allan H. Kopp
Ada Joint Program Office

Dr. Richard Kuntz
Monmouth College

Dr. Ronald Leach
Howard University

Dr. Susan Richman
The Pennsylvania State
University at Harrisburg

John W. Roberts
The BDM Corp.

Albert Rodriguez
HQ, CECOM

Walter Rolling
AdaSoft, Inc.

Susan Rosenberg
SofTech, Inc.

Jerry Rudisin
Alsys, Inc.

Ruth Rudolph
Computer Science Corp.

Michael Sapenter
TEOLOS Federal Systems

MAJ Doug Samuels
HQ AFSC/PLRT

Jeffrey E. Simo.ı
MA/COM Government
Systems

Terrence P. Starr
General Electric Co.

Kay Trezza
HQ, CECOM

James E. Walker
Network Solutions

This conference's

PANELS AND TECHNICAL SESSIONS

dealt with these topics:

Tuesday, March 15, 1988

9:00 AM	Opening Reception	
9:00 AM	Panel I	Ada in the Life Cycle;
2:15 PM	Session 1	Metrics,
2:15 PM	Session 2	Life Cycle Environments,
2:15 PM	Session 3	Distributed Systems.

Wednesday, March 16, 1988

8:30 AM	Panel II	STARS Technology Update,
10:45 AM	Session 4	Ada Training and Education;
10:45 AM	Session 5	Ada Applications;
10:45 AM	Session 6	Ada Technology and Research,
2:15 PM	Session 7	~~Ada Training and Education II~~
2:15 PM	Session 8	~~Ada Applications II~~
2:15 PM	Session 9	~~Ada Technology and Research II~~

Thursday, March 17, 1988

8:45 AM	Session 10	Life Cycle Management,
8:45 AM	Session 11	Reusability,
8:45 AM	Session 12	~~Ada Application III~~
8:45 AM	Session 13	Ada and Systems Design;
1:00 PM	Session 14	~~Life Cycle Management II~~
1:00 PM	Session 15	Realtime Systems,
1:00 PM	Session 16	~~Ada Application IV~~
1:00 PM	Session 17	Ada ~~and Systems Design II~~
2:45 PM	Panel III	Looking to the Future with Ada .

PAPERS

The papers in this volume were printed directly from unedited reproducible copies prepared by the authors. Responsibility for contents rests upon the authors and not the symposium committee or its members. After the symposium, all the publication rights of each paper are reserved by their authors, and requests for republication of a paper should be addressed to the appropriate author. Abstracting is permitted, and it would be appreciated if the symposium is credited when abstracts or papers are republished. Requests for individual copies of papers should be addressed to the authors.

PROCEEDINGS

SIXTH NATIONAL CONFERENCE ON ADA TECHNOLOGY

Bound—Available at Fort Monmouth

2nd Annual National Conference on Ada Technology Proceedings—1984—$10.00
3rd Annual National Conference on Ada Technology Proceedings—1985—$15.00
4th Annual National Conference on Ada Technology Proceedings—1986—$20.00
5th Annual National Conference on Ada Technology Proceedings—1987—(Not Available)
*6th Annual National Conference on Ada Technology Proceedings—1988—$25.00

*Extra copies: 1-3 $25.00; next 4-10 $20; next 11 & above $15.00 each

Make check or bank draft payable in U.S. dollars to the Annual National Conference on Ada Technology and forward request to:

Annual National Conference on Ada Technology
U.S. Army Communications—Electronics Command
ATTN: AMSEL-RD-SE-CRM (Ms. Kay Trezza)
Fort Monmouth, New Jersey 07703-5000

Telephone inquiries may be directed to Ms. Kay Trezza at (201) 532-1898.

Photocopies—Available at Department of Commerce. Information on prices and shipping charges should be requested from the:

U.S. Department of Commerce
National Technical Information Service
Springfield, Virginia 22151
USA

Include title, year, and AD Number

2nd Annual National Conference on Ada Technology—1984-AD A142403
3rd Annual National Conference on Ada Technology—1985-AD A164338
4th Annual National Conference on Ada Technology—1986-AD A167802
5th Annual National Conference on Ada Technology—1987-AD A178690

Highlights
Joint Ada Conference
Fifth Annual National Conference on Ada Technology
March 16-19, 1987
Crystal Gateway Marriott Hotel, Arlington, Virginia

Greetings

Mr. James E. Schell, U.S. Army, CECOM, Ft. Monmouth, NJ

The Honorable Robert C. Byrd, Senator of West Virginia

General Speakers

LTG Thuman D. Rogers, Assistant Chief of Staff for Information Management, U.S. Army, Washington, D.C.

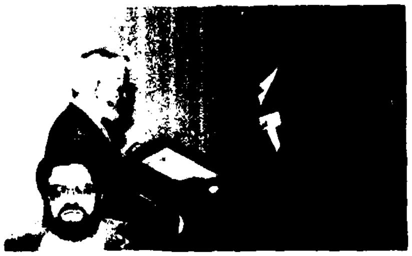
Honorable Donald C. Latham, Assistant Secretary of Defense (Command, Control, Communications and Intelligence), Washington, D.C.

Rear Admiral Harry S. Quast, Director, Dept of Navy, Information Resources Management, Washington, D.C.

Dr. Andrew Stofan, NASA Associate Administrator, Space Station, NASA Headquarters, Washington, D.C.

Special Recognition Awards

Mr. James Schell, U.S. Army, CECOM, Ft. Monmouth, NJ presenting Awards to:

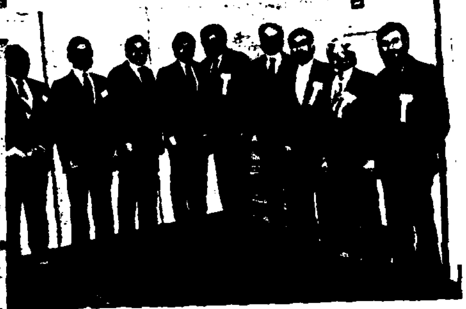


Mr. John Roberts, BDM Corp—Committee Chairman 1986-1987

Ms. Charlene Hayden, GTE Government Systems—Committee Treasurer 1985-1987

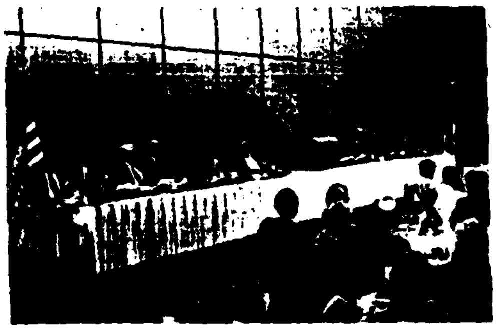
Mr. Albert Rodriguez, U.S. Army CECOM, Ft. Monmouth, NJ—Program Committee

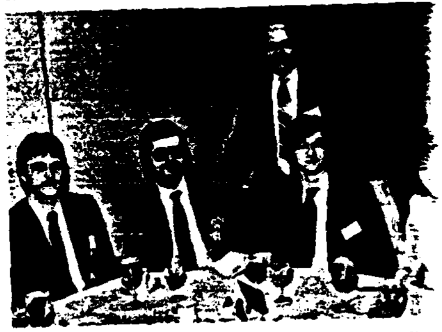
Mr. Daniel Roy, Century Computing—Program Committee

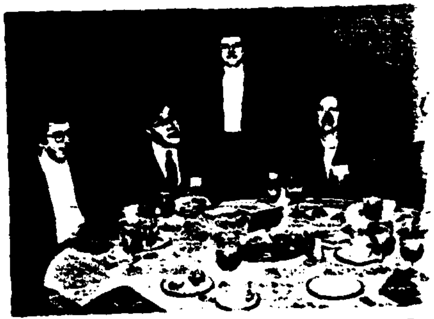


Academia Government Industry Ada

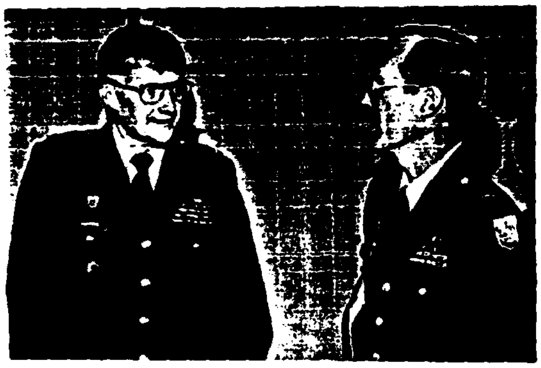
SPEAKERS BREAKFAST







Marriott
Crystal Gateway
Academia
Government
Ada
Industry
REGISTRATION



Ada Conference

MARCH 16 - 19, 1987

Academia Government Ada Industry

Ada Conference

MARCH 16 - 19, 1987

Academia Government Ada Industry

TUTORIALS

Academia
Government
Ada
Industry
Panel
ning
TUTORIALS

TABLE OF CONTENTS

TUESDAY, MARCH 15, 1988—9:00 AM-12:00 N

Arlington Ballroom—Salon IV

OPENING RECEPTION:

Moderator:

Mr. James E. Schell, Director, Center for Software Engineering, U.S. Army Communications Electronics Command, Fort Monmouth, NJ

Greetings:

Dr. Jesse C. Lewis, Vice President for Academic Affairs, Norfolk State University, Norfolk, VA

Dr. Joseph E. Gilmour, Executive Assistant to the Chancellor, University of Maryland, College Park, MD

PANEL DISCUSSION I

Ada in the Life Cycle

Chairperson: Mr. James E. Schell, Center for Software Engineering, U.S. Army Communications Electronics Command, Fort Monmouth, NJ

Panelists:

MG David Teal, Deputy Chief of Staff, Product Assurance Acquisition Logistics, U.S. Air Force Systems Command, Andrews AFB, Washington, DC

BG Alonzo Short, Commanding General, U.S. Army Information Systems Engineering Command, Ft. Belvoir, VA

Rear Admiral Harry S. Quast, Director, Information Systems Division, Department of the Navy, Washington, DC

BG Richard L. Phillips, Director, Command, Control, Communications and Computer (C4) Systems Division, HQ, U.S. Marine Corps, Washington, DC

Mr. Loren Diedrichsen, Principal Technical Advisor, NATO Communications and Information Systems Agency (NACISA), Brussels

Ms. Virginia Castor, Director, Ada Joint Program Office, Arlington, VA

Dr. Larry Druffel, Director, Software Engineering Institute, Carnegie Mellon Institute, Pittsburgh, PA

LUNCH, 12:10 N-2:00 PM

Arlington Ballroom—Salons I, II and III

Guest Speaker: Major General Billy M. Thomas, Commanding General, U.S. Army Communications-Electronics Command, Ft. Monmouth, NJ

TUESDAY, MARCH 15, 1988—2:15 PM-5:30 PM

Arlington Ballroom—Salon I and II

SESSION 1: METRICS

Chairperson: Mr. Daniel Hocking, AIRMICS, Atlanta, GA

Arlington Ballroom—Salon IV

SESSION 2: LIFE CYCLE ENVIRONMENTS

Chairperson: Mr. Albert Rodriguez, U.S. Army, CECOM, Ft. Monmouth, NJ

Arlington Ballroom—Salons V and VI

SESSION 3: DISTRIBUTED SYSTEMS

Chairperson: Dr. Ronald Leach, Howard University, Washington, DC

TUESDAY, BANQUET 7:00 PM-9:00 PM

Arlington Ballroom—Section III and IV

WEDNESDAY, MARCH 16, 1988—8:30 AM-10:30 AM

Arlington Ballroom—Salon IV

PANEL DISCUSSION II: STARS TECHNOLOGY UPDATE

Chairperson: Col. Joseph Green, U.S. Air Force, Director of STARS Program, Washington, DC

Panelists:

Richard E. Barbour, Jr., US Navy, Deputy Director of STARS Program, Washington, DC

Glenn E. Hughes, II, Cpt., U.S. Army, Office of Director of Information Systems for C4, Washington, DC

Rebecca G. Abraham, Cpt., U.S. Air Force, WPAFB, Ohio.

Geraldine R. Cuthbert, Software Technologies, Melbourne, FL

William B. Easton, Peregine Systems, Inc., Consultant to Institute for Defense Analysis, Alexandria, VA

Robert S. Cymbalski, Science Applications International Corp. (SAIC), Clearwater, FL

WEDNESDAY, MARCH 16, 1988—10:45 AM-12:00 PM

Arlington Ballroom—Salon I and II

SESSION 4: ADA TRAINING AND EDUCATION I

Chairperson: Dr. M. Susan Richman, The Pennsylvania State University at Harrisburg, Middletown, PA

Arlington Ballroom—Salon III

SESSION 5: ADA APPLICATIONS I

Chairperson: Mr. James Walker, Network Solutions, Vienna, VA.

Arlington Ballroom—Salon V and VI

SESSION 6: ADA TECHNOLOGY AND RESEARCH I

Chairperson: Ms. Susan Rosenberg, SofTech Inc., Waltham, MA

LUNCH, 12:10 PM-2:00 PM

Arlington Ballroom—Salon III and IV

Guest Speaker: Ltg. Emmett G. Paige Jr., Commander, U.S. Army Information Systems Command, Ft. Huachuca, AZ

WEDNESDAY, MARCH 16—2:15 PM-5:30 PM

Arlington Ballroom—Salon I and II

SESSION 7: ADA TRAINING AND EDUCATION II

Chairperson: Ms. Ruth Rudolph, Computer Science Corp., Moorestown, NJ

Arlington Ballroom—Salon III

SESSION 8: ADA APPLICATIONS II

Chairperson: Ms. Dee Graumann, General Dynamics, DSD, San Diego, CA

Arlington Ballroom—Salon V and VI

SESSION 9: ADA TECHNOLOGY AND RESEARCH II

Chairperson: Dr. Arthur M. Jones, Morehouse College, Atlanta, GA

WEDNESDAY, BANQUET—7:00 PM-9:00 PM

Arlington Ballroom—Salon III and IV

THURSDAY, MARCH 17, 1988—8:45 AM-12:00 N

Arlington Ballroom—Salon I and II

SESSION 10: LIFE CYCLE MANAGEMENT I

Chairperson: Major Doug Samuels—Andrews AFB, MD

Arlington Ballroom—Salon IV

SESSION 11: REUSABILITY

Chairperson: Mr. John Roberts, The BDM Corp., Norfolk, VA

Arlington Ballroom—Salon III

SESSION 12: ADA APPLICATION III

Chairperson: 1LT Shelia Bryant, Marine Corps Tactical Systems Support Activity, Camp Pendleton, CA

Arlington Ballroom—Salon V and VI

SESSION 13: ADA AND SYSTEMS DESIGN I

Chairperson: Mr. Michael Sapenter, Telos Federal Systems, Lawton, OK

THURSDAY, MARCH 17—1:00 PM-2:30 PM

Arlington Ballroom—Salon I and II

SESSION 14: LIFE CYCLE MANAGEMENT II

Chairperson: Dr. Richard Kuntz, Vice President, Research and Technology, Monmouth College, West Long Branch, NJ

Arlington Ballroom—Salon IV

SESSION 15: REALTIME SYSTEMS

Chairperson: Mr. Brian Baker, Navy Department, Washington, DC

Arlington Ballroom—Salon III

SESSION 16: ADA APPLICATION IV

Chairperson: Mr. James Walker, Network Solutions, Vienna, VA

Arlington Ballroom—Salon V and VI

SESSION 17: ADA AND SYSTEMS DESIGN II

Chairperson: Mr. Walter Rolling, AdaSoft, Inc., Lanham, MD

THURSDAY, MARCH 17, 1988—2:45 PM-4:15 PM

Arlington Ballroom—Salon III and IV

PANEL DISCUSSION III: FUTURE DIRECTIONS OF ADA

Chairperson: Miguel A. Carrio, Jr., Teledyne Brown Engineering, Fairfax, VA

Panelists:

Dr. Laszlo A. Belady, Microelectronics and Computer Technology Corporation, Austin, TX

Dr. Barry Boehm, TRW, Inc., Redondo Beach, CA

Dr. William Riddle, Software Productivity Consortium, Reston, VA

Dr. Jean Ichbiah, Alsys, Inc., Waltham, MA

TestGen - Testing tool for Ada Designs and Ada Code.

Thomas S. Radi, Ph.D.

Software Systems Design, Inc.

This paper describes a software program, TestGen, that assists in the testing of executable Ada code as well as assisting in the testing of high level descriptions of Ada designs.

The TestGen program provides three distinct capabilities:
1. The Design Review Expert Assistant-
   Allows Ada designs to be thoroughly reviewed, insuring that all paths have been evaluated, and that all possibilities have been covered.
2. The Unit Test Strategy Generator-
   Assists in the definition of unit test procedures using a "white box" testing technique.
3. The Test Coverage Analyzer-
   Determines the extent of coverage (the percentage of the total numbers of paths, branches and statements that were actually executed during a given test sequence).

The TestGen tool is one of the AISLE (Ada Integrated Software Lifecycle Environment) toolset, an integrated set of tools that assist the developers of Ada software.

Introduction

This paper describes TestGen, a tool which was developed to assist in the review and testing of Ada code.

TestGen is one of an integrated family of tools which assist in the design, coding, documentation and testing of Ada programs.

The utility of TestGen is in four primary areas:

1. TestGen can insure that Design Reviews are complete, and that every possible set of conditions is examined to determine the proposed consequences of that set of conditions.

2. TestGen can be used in conjunction with a "white box" testing approach to prepare Unit Test plans and procedures for each program unit.

3. TestGen can be used to estimate the testing complexity, by determining the number of tests required to insure complete path coverage, or branch coverage or some combination of approaches.

4. TestGen can be used in conjunction with a "black box" testing approach to determine the effectiveness of a set of tests, i.e. to determine the number of paths or branches that a series of tests has executed.

The paper briefly describes the TestGen tool, shows some examples of the TestGen output reports and describes how the tool is being used.

Control Structure Determines Testing Effort

Executable Ada source code and Ada/PDL designs which are expressed using a high level structured pseudo-code have an internal control structure that can be analyzed to determine the total number of paths through a program unit.

These paths form the "graph" of the control logic associated with each program unit in an Ada program.

By examining the conditions at each branch point, as expressed in pseudo-code for the Ada design, and as expressed in executable code, two independent control graphs are developed to describe the control structure of the design and the code respectively.

Figure 1 shows an example of some Ada code, and the associated control graphs.

The nodes of the control graphs represent branch points (typically conditions) in the Ada design or in the Ada program. The paths connecting the nodes consist of statements (pseudo-code or executable code) in the Ada program.

During the creation of the control graphs, each node is associated with the statements from the original Ada source file that correspond to the condition at the node. These statements are used to textually identify the conditions that are required at each branch point and also identifies the statements that will be executed as a result of the conditions.

Once the control graph is known for both the design and the code, the graph (as stored internally by the TestGen program) is used to determine
- a) the number of branches through the program
- b) the number of paths through the program
- c) the cyclomatic number (for McCabe Structured Testing)

The number of paths, branches and cyclomatic number provide the user with an estimate of the time required to use either of three white box testing methodologies:

a) total branch coverage - where all conditions at each branch point are executed at least once.

b) total path coverage - where every possible path through the design/program is covered.

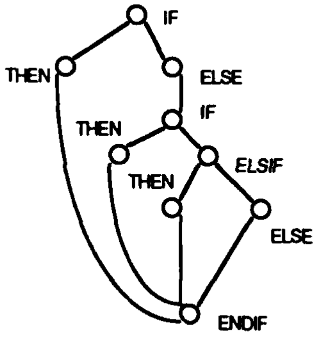
c) Cyclomatic (structured) testing - a midground between branch and path coverage where the cyclomatic number is used to determine the number of tests required.

Figure 1. An example of a control graph

The code fragment

```
procedure A_dog_of_an_example is
--¦ if the dog is hungry then
--¦  feed it
--¦ else
--¦  take its temperature
--¦   if it has a fever then
--¦    take it to the vet
--¦   elsif it has a dry nose
--¦    give it some water
--¦   else
--¦    wipe its nose
--¦   end if
--¦
begin
null;  --  executable code not shown
--
end A_dog_of_an_example;
```

The control graph



TestGen works in conjunction with the ADADL Ada/PDL processor

ADADL is the Ada-based Design And Documentation Language. The ADADL processor analyzes designs described using Ada/ADADL to produce reports that highlight various aspects of the design. When used in conjunction with TestGen, the ADADL processor analyzes the Ada source code to determine the control structure of each program unit.

During the analysis of the source file, the ADADL processor creates an intermediate control structure file, in which every statement is identified either as a passive statement (not containing any branch conditions), or as a statement containing a possible branch condition. The specific condition at each branch point is identified and associated textually with the branch point.

TestGen Design Review Expert Assistant identifies all possible conditions.

If the designer has used an Ada/PDL pseudo-code to describe the design algorithm, the Design Review Expert Assistant (DREA) portion of TestGen can be used to insure that a thorough design review is conducted. The DREA identifies all possible conditions in each program unit, and delineates the expected result for each set of conditions.

Figure 2 is the ADADL pretty print output of the example in Figure 1. There are four possible paths through the design. Figure 3 is the output of the TestGen Design Review Expert Assistant analysis of the example.

Figure 2. The Example of figure 1 with ADADL Line Numbers.

```
LINE
1  procedure A_dog_of_an_example is
2    --¦ if the dog is hungry then
3    --¦   feed it
4    --¦ else
5    --¦  take its temperature
6    --¦   if it has a fever then
7    --¦    take it to the vet
8    --¦   elsif it has a dry nose
9    --¦    give it some water
10   --¦   else
11   --¦    wipe its nose
12   --¦   end if
13   --¦
begin
null;  --  executable code not shown
--
end A_dog_of_an_example;
```

Figure 3. A fragment of the Design Review Expert Assistant output.

```
*******************************************
 Reviewing all paths of  Subprogram:
A_Dog_Of_An_Example
*******************************************
-------------------------------------------
 Design conditions reviewed case 1 of 4 for
subprogram: A_Dog_Of_An_Example

The Design conditions examined for design case 1 are:

 2:  ( the dog is hungry ) is False
 6:  ( it has a fever ) is False
 8:  ( it has a dry nose) is False

Expected results for design case 1 are:

 5:   take its temperature
 11:   wipe its nose

-------------------------------------------
 Design conditions reviewed case 2 of 4 for
subprogram: A_Dog_Of_An_Example

The Design conditions examined for design case 2 are:

 2:  ( the dog is hungry ) is False
 6:  ( it has a fever ) is False
 8:  ( it has a dry nose) is True

Expected results for design case 2 are:

 5:   take its temperature
 9:   give it some water

-------------------------------------------
 Design conditions reviewed case 3 of 4 for
subprogram: A_Dog_Of_An_Example

The Design conditions examined for design case 3 are:

 2:  ( the dog is hungry ) is False
 6:  ( it has a fever ) is True

Expected results for design case 3 are:

 5:   take its temperature
 7:   take it to the vet

-------------------------------------------
 Design conditions reviewed case 4 of 4 for
subprogram: A_Dog_Of_An_Example

The Design conditions examined for design case 4 are:

 2:  ( the dog is hungry ) is True

Expected results for design case 4 are:

 3:   feed it
```

Since the DREA shows the review team every possible path through the proposed design, using the DREA helps insure that all design decisions which affect the flow of control through the program have been examined.

Unit Test Strategy Generator

The Unit Test Strategy Generator (TSG) is used to determine the test conditions that a test engineer will need to set up to insure the complete testing of the executable Ada code.

The TSG supports three "white box" testing methodologies: complete path testing, branch coverage testing and Structured (McCabe) Testing as identified in NBS Publication 500-99 (National Bureau of Standards).

The executable Ada code is analyzed by the ADADL processor to determine all of the relevant control/decision points.

The user can specify the methodology of testing that he/she wishes to employ during the unit test for each module.

The TSG will subsequently identify the conditions that must be set up at each branch point to insure that all necessary tests are run according to the testing methodology specified (total path coverage, branch coverage or Structured Testing).

Note that there is no guarantee that the conditions identified by TSG are feasible conditions. Indeed the program itself may very well prevent the execution of certain paths as shown below. There are four possible paths through the example. The path where a is true and c is true will never be executed. The path where a is a is false and c is false can never be executed.

```
procedure a is
a,b,c:boolean := false ;
if a then
b:=true;
else
c:=true;
end if;
if c then
c_true;
else
c_false;
end if;
```

One observation at this point is that perhaps the unit under test should be examined for possible elimination of infeasible conditions.

The example above could be re-written as

```
procedure a is
a,b,c:boolean := false ;
if a then
b:=true;
c_false;
else
c:=true;
C_true;
end if;
```

The TSG helps identify infeasible paths

We claim that by (human) examination of the TSG strategy, infeasible condition n-tuples (which are paths in the program) can be identified.

We do not claim that all of these infeasible paths may always be eliminated by suitable re-coding of the algorithm. However, the identification of these infeasible paths may lead the test engineer to a closer examination of these "anomalies", which may indeed lead to a more reasonable design.

The TSG provides three test strategy options:

1. Complete Path Testing
With complete path testing, the TSG identifies the conditions at every possible control point in the unit under test, to insure that every path in the program is executed at least once during a series of tests.

2. Branch Coverage
With branch coverage, the TSG identifies the conditions at each control point to insure that each possible branch is exercised with the branch point taking on every possible value. Branch coverage is equivalent to insuring that every (reachable) statement has been executed.

3. Structured Testing
Structured Testing, as identified by McCabe in National Bureau of Standards Publication 500-99, is a combination of Branch coverage and Path coverage. If the user selects Structured Testing, TSG identifies the conditions at each branch point to insure that the requisite number of tests will be run. In structured testing the cyclomatic number (McCabe metric) identifies the suggested number of tests. There are usually several possible sets of Structured Tests that can be identified for a given unit under test; the TSG identifies one set of tests.

Test Coverage Analyzer Determines the Extent of Functional Test Success.

The TestGen Test Coverage Analyzer (TCA) is used to determine the effectiveness of a set of tests. The TCA may be used either during "white box" Unit Test, or during functional "black box" testing of the program.

The TCA consists of two parts:

Ada Code Test Coverage Instrumenter

Test Coverage Profiler

The Ada Code Instrumenter Instruments the Code to be tested.

The TCA asks the user to identify two things:
(a) the list of program units to be tested during a particular run (this identifies the units under test), and (b) the type of testing desired, where the user will identify whether he wishes to instrument the units under test for Branch coverage or path coverage.

The Ada code Test Coverage Instrumenter will place "instrumentation code" in the appropriate places in the list of units under test . The ACI attempts to minimize the number and amount of instrumentation code that is placed in the code to be tested.

The user must recompile the "instrumented" Ada source code, and run the tests using the results of the compilation of the instrumented code.

The Test Coverage Profiler quantifies the effectiveness of test coverage.

The TestGen Test Coverage Profiler (TCP) quantifies the effectiveness of the series of tests which have been run against the units under test.

The TCP generates reports which show the user how effective a particular series of tests have been with respect to the execution of all of the branches and/or paths.

Of equal interest to the test engineer is an identification of those branches and/or paths which have not been executed during the testing process. Knowing the set of statements which have not been executed, and using TestGen's Unit Test Strategy Generator, the test engineer can easily prepare additional test conditions that will execute the missing branches and paths, or the test engineer can determine why the particular branches and paths were not executed as originally expected.

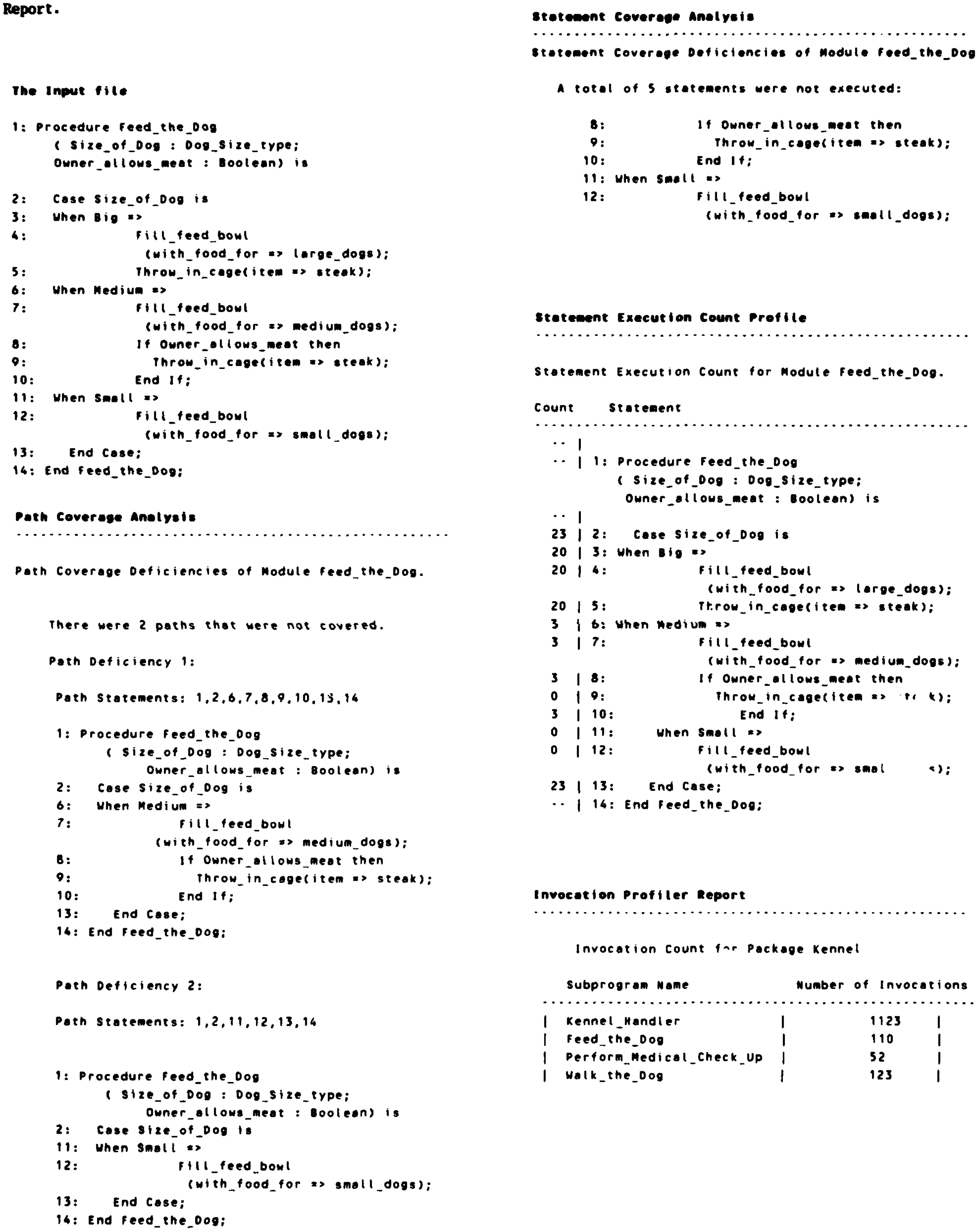
Test Coverage Profiler also provides Timing Estimates

In addition to delineating the sets of branches and paths which have been executed during a test, the TCP can provide estimates as to the percentage of time the program has spent in each routine.

This capability is useful in determining where any additional optimization of the code would have the most effect in terms of execution speed.

A typical output report is shown in Figure 4.

Figure 4. A portion of the Test Coverage Profiler Report.



```
The Input file

1: Procedure Feed_the_Dog
     ( Size_of_Dog : Dog_Size_type;
     Owner_allows_meat : Boolean) is

2:   Case Size_of_Dog is
3:   When Big =>
4:             Fill_feed_bowl
               (with_food_for => large_dogs);
5:             Throw_in_cage(item => steak);
6:   When Medium =>
7:             Fill_feed_bowl
               (with_food_for => medium_dogs);
8:             If Owner_allows_meat then
9:               Throw_in_cage(item => steak);
10:            End If;
11:  When Small =>
12:            Fill_feed_bowl
               (with_food_for => small_dogs);
13:    End Case;
14: End Feed_the_Dog;

Path Coverage Analysis
-----------------------------------------------------

Path Coverage Deficiencies of Module Feed_the_Dog.


    There were 2 paths that were not covered.

    Path Deficiency 1:

     Path Statements: 1,2,6,7,8,9,10,13,14

     1: Procedure Feed_the_Dog
           ( Size_of_Dog : Dog_Size_type;
                Owner_allows_meat : Boolean) is
     2:   Case Size_of_Dog is
     6:   When Medium =>
     7:             Fill_feed_bowl
               (with_food_for => medium_dogs);
     8:             If Owner_allows_meat then
     9:               Throw_in_cage(item => steak);
     10:            End If;
     13:    End Case;
     14: End Feed_the_Dog;


     Path Deficiency 2:

     Path Statements: 1,2,11,12,13,14


     1: Procedure Feed_the_Dog
           ( Size_of_Dog : Dog_Size_type;
                Owner_allows_meat : Boolean) is
     2:   Case Size_of_Dog is
     11:  When Small =>
     12:            Fill_feed_bowl
                    (with_food_for => small_dogs);
     13:    End Case;
     14: End Feed_the_Dog;
```

```
Statement Coverage Analysis
-----------------------------------------------------
Statement Coverage Deficiencies of Module Feed_the_Dog

   A total of 5 statements were not executed:

      8:             If Owner_allows_meat then
      9:               Throw_in_cage(item => steak);
     10:             End If;
     11: When Small =>
     12:             Fill_feed_bowl
                      (with_food_for => small_dogs);


Statement Execution Count Profile
-----------------------------------------------------

Statement Execution Count for Module Feed_the_Dog.

Count    Statement
-----------------------------------------------------
  -- |
  -- | 1: Procedure Feed_the_Dog
          ( Size_of_Dog : Dog_Size_type;
           Owner_allows_meat : Boolean) is
  -- |
 23  | 2:   Case Size_of_Dog is
 20  | 3: When Big =>
 20  | 4:            Fill_feed_bowl
                      (with_food_for => large_dogs);
 20  | 5:            Throw_in_cage(item => steak);
 3   | 6: When Medium =>
 3   | 7:            Fill_feed_bowl
                      (with_food_for => medium_dogs);
 3   | 8:            If Owner_allows_meat then
 0   | 9:              Throw_in_cage(item =>  tr  k);
 3   | 10:                End If;
 0   | 11:      When Small =>
 0   | 12:           Fill_feed_bowl
                      (with_food_for => smal     <);
 23  | 13:    End Case;
  -- | 14: End Feed_the_Dog;



Invocation Profiler Report
-----------------------------------------------------

    Invocation Count f^r Package Kennel

   Subprogram Name              Number of Invocations
-----------------------------------------------------
 |  Kennel_Handler            |         1123     |
 |  Feed_the_Dog              |         110      |
 |  Perform_Medical_Check_Up  |         52       |
 |  Walk_the_Dog              |         123      |
```

Future Efforts

The TestGen tools are currently operational on several computer systems, including VAX/VMS VAX/Unix, Sun, Apollo, Data General and Harris.

The tools are currently configured are designed to be totally independent of the Ada compiler being used on the host and/or target system.

A closer integration of TestGen with a compiler's debugger is an effort that is planned for the future. This integration will provide the additional capability of running "uninstrumented" Ada code on the target machine. A subsequent analysis of the execution paths on the target would determine the branches and paths that were (and were not) executed.

Conclusion

We have described the capabilities of TestGen, a tool which was developed to assist in the Testing of Ada designs (as expressed in a structured pseudo-code), and in the testing of actual Ada code.

Testing encompasses all phases of the development lifecycle. During the Design phase, the testing process is a process of Design Review. The Design Review Expert Assistant portion of the TestGen tool assists in insuring that the design of each unit id thoroughly reviewed.

The Unit Test Strategy Generator assists in the preparation of the test plans and procedures for each program unit. The number of tests required is determined by the complexity of the unit being tested, and the test methodology selected.

The Test Coverage Analyzer reports on the effectiveness of a series of tests which are performed on the program.

Author Biography:

Dr. Radi is president of Software Systems Design, Inc. He has over 15 years of experience in the Aerospace/Electronics industry in both Hardware design (logic and computer design) and Software development. He is the Software Design Manager at the General Dynamics/Pomona Division. He is the originator of ADADL one of the most popular Ada/PDL processors, and the AISLE (Ada Integrated Software Lifecycle Environment) family of Ada software development tools.

Ada Complexity Extension (ACE)
An extension of McCabe's Cyclomatic Complexity Metric for Analysis of Ada Software

Holly J. Tauson-Conte

TELEDYNE BROWN ENGINEERING

ABSTRACT

This article contains the results of initial research work performed to extend the applicability of McCabe's Cyclomatic Complexity Metric for the analysis of Ada software. Having proved useful both as a logical measurement technique and as a testing aid, the Ada Complexity Extension (ACE) is proposed for general acceptance as a standard to provide a useful metric that may assist in improving the quality of Ada software programs.

1. INTRODUCTION

Ada is designed to support modern software engineering principles. The use of Ada-oriented metrics, together with the features of the Ada language, constitute a valuable framework against which to apply these principles during software development. To address the need for Ada metrics, this paper proposes a metric, Ada Complexity Extension (ACE), which describes, quantifies, and makes visible the logical complexity of Ada modules [1]. At the time of this publication this theory has not been validated by extensive empirical data.

The ACE metric is an extension of McCabe's Cyclomatic Complexity Metric and preserves many of its basic properties. ACE has a mathematical basis in graph theory, supports structured programming, and is applicable to multiple phases in the software life cycle.

2. HISTORICAL BACKGROUND

McCabe developed and published in the 1970's a graph-theoretic complexity measure to address the need for a mathematical technique that provides a quantification of software system modules [3]. The technique is intended to assist in the identification of software modules that may consume time and prove costly during test and maintenance.

The properties of McCabe's Cyclomatic Complexity Measure include its mathematical basis, its support of structured programming, its applicability to multiple phases of the software life cycle (design, implementation, testing [4], and maintenance), and empirical support through case studies verifying correlation between logical complexity and software program errors. This complexity theory is pragmatic because it assists with the management of software complexity by making control-flow visible and quantifiable. Each software module may be depicted as a flow graph by associating a block of sequential code with a graph node and a program logical transfer of control with an arc. Applying graph theory to a flow graph of a program module yields a cyclomatic complexity number by evaluating the formula

$$v(G) = e - n + 2,$$

where e is the number of edges and n is the number of nodes.

At the time of McCabe's complexity theory's introduction in 1976, high-order programming languages were using expression and control abstraction as initial mechanisms for controlling complexity. The application of McCabe's complexity measure to the expression, control, and procedural abstraction of languages in the mid-1970's was a relatively direct mapping. Ada's advanced language capabilities, such as strong typing, packages, tasks, generic units, and exception handling, go beyond the capabilities of most preceding general-purpose high-order languages and therefore generate an impact on the traditional McCabe complexity metric. The ACE theory incorporates evaluation of data abstraction (packages) and process abstraction (tasks), plus the remaining special features present in the Ada language, to provide an accurate static evaluation of Ada modules.

3. MAJOR DIFFERENCES BETWEEN ACE AND McCABE'S METRIC

It was necessary to modify and extend the traditional McCabe metric for complexity analysis of Ada modules because the Ada language contains a number of constructs and statements that may be used for inter-module dependency and communication and other unique language features. The ACE theory evaluates complexity by applying the formula (e-n+2) to flow graphs that have been modified to appropriately represent the flow of Ada program modules. The major modifications are discussed briefly in this section, to be followed by a more detailed description in Section 4.

The static complexity analysis of program modules that involve inter-module dependency and communication may result in interrupted program module flow paths. The interrupted paths may be illustrated on the flow graph by unconnected nodes and, in some cases, specialized arcs connected to a single node are introduced to depict either a dynamic transfer into the module or a dynamic transfer out of the module.

Ada package bodies and generic units require further extensions of McCabe's metric. Complexity evaluation of Ada package bodies requires a recognition of the optional sequence of Ada statements that may be present in the executable part of the package body as well as the nested bodies that are contained in the declarative part of the package body. The complexity analysis of nongeneric units produced by the Ada compiler as a result of instantiations of the generic units cannot be acquired through source code analysis. The complexity value of each instance of a nongeneric unit must relate back to its corresponding generic unit code analysis.

The ACE does not attempt to measure the dynamic features, such as creation, termination, and suspension of Ada tasks, but does include the static complexity analysis of Ada features, such as the select statement, guard conditions, the abort statement, and entry calls.

The Ada exception handler feature is yet another reason to extend McCabe's traditional metric. An Ada exception handler represents replacement or recovery code, which may incorporate a large variety of Ada constructs resulting in a recovery algorithm that may be as complicated or more complex than the abandoned algorithm. The ACE theory introduces a composite value to quantify the complexity of an Ada program module that contains one or more exception handlers.

4. ACE DESCRIPTION

The traditional McCabe metric is designed to ane'yze modular divisions of a larger program. The definition of a program module varies from programming language to programming language and it is appropriate to identify those units that constitute analyzable Ada program modules. Certain Ada compilation units and certain declarative items are the essential units that may have a static complexity value associated with them. By eliminating the compilation units that contain only declarations, the analyzable Ada program modules include subprogram bodies, package bodies, generic bodies, and subunits. In addition, nested declarations of proper bodies are also considered analyzable program modules. These nested declarations include subprogram bodies, package bodies, task bodies, and generic unit bodies.

The static evaluaton of Ada constructs and features according to the ACE theory is described in the following subsections. Flow graphs illustrate the path analyses associated with example Ada program modules.

4.1 Ada Conditional Statements

The first kind of conditional construct, the Ada if statement, has the same complexity as that described in McCabe's metric. The second conditional construct, the Ada case statement, selects one of a number of alternative sequences of statements for execution, depending on the value of an expression. Since the alternative choices are exhaustive and mutually exclusive, the Ada case statement with x alternatives has a complexity measure of x - 1.

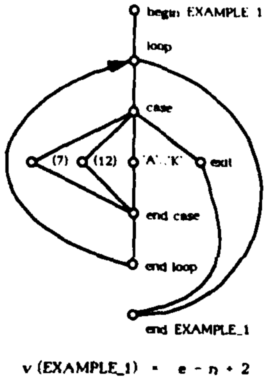
4.2 Ada Loop Statements

The Ada loop statement may be expressed in a variety of ways, with the completion of the loop execution depending on an iteration scheme, the execution of an exit statement, or some other transfer of control. The Ada basic loop statement involves no evaluation of a logical decision and therefore does not contribute to the complexity of the program module. Unlike the basic loop, the Ada "while" and "for" loops both contain an iteration scheme that represents a logical decision regarding completion of the loop. Therefore, the evaluation of an Ada loop with an iteration scheme will result in contributing one to the value of the module complexity.

Any form of the Ada loop statement may contain an exit statement. If the exit statement includes a condition, the loop will complete when the exit statement is reached and the condition evaluates to TRUE. The exit statement that does not include a condition will cause the loop to complete when the exit statement is reached. The conditional exit statement then will contribute one, whereas the unconditional exit statement will not contribute to the program module complexity. The complexity evaluation of an Ada procedure that contains a loop plus a case statement is illustrated in Graph 1. The Ada procedure is given in Figure 1.

FIGURE 1

```
procedure EXAMPLE_1 is
. . .
begin
   OPEN (FILE, . . .);
   while not END_OF_FILE (FILE) loop
      TEXT_IO.GET ( INPUT_CHAR );
      case INPUT_CHAR is
         when Character'VAL (7)   => RING_BELL;
         when Character'VAL (12)  => SKIP_TO_NEXT_PAGE_POSITION;
         when 'A' . . 'K'         => CHOOSE_A_MENU;
         when others              => exit;
      end case;
   end loop;
   CLOSE (FILE);
end EXAMPLE_1;
```

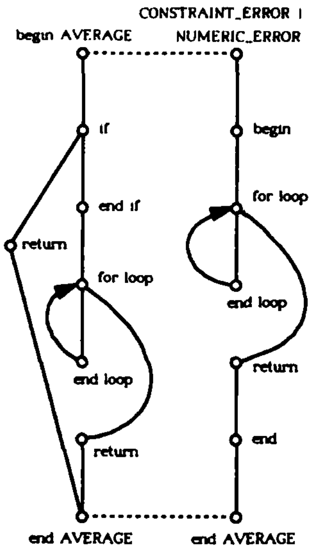


$$\begin{aligned} v\,(\text{EXAMPLE\_1}) &= e - n + 2 \\ &= 13 - 10 + 2 \\ &= 5 \end{aligned}$$

Graph 1

FIGURE 2

```
type Vector is array (Integer range <>) of Integer;
. . .
function AVERAGE ( V : Vector) return Float is
   SUM : Integer := 0;
begin
   if V'LENGTH = 0 then
      return 0.0;
   end if;
   for J in V'RANGE loop
      SUM := SUM + V (J);
   end loop;
   return Float (SUM) / Float (V'LENGTH);
exception
   when CONSTRAINT_ERROR | NUMERIC_ERROR =>
      declare
        SUM : Float := 0.0;
      begin
        for J in V'RANGE loop
          SUM := SUM + Float ( V(J) );
        end loop;
        return  SUM / Float (V'LENGTH);
     end;
end AVERAGE;
```



$$v\,(\text{AVERAGE}) = (e-n+2,\ e-n+2)$$
$$= (9-8+2,\ 7-7+2)$$
$$= (3,\ 2)$$

Graph 2

4.3 Ada Exception Handlers

During program execution, an Ada exception may arise from several sources: a raise statement, another Ada statement, an operation that propagates the exception, or during the elaboration of declarations. When an exception is raised, control transfers to a user-provided exception handler, which may occur at the end of a block statement or at the end of the body of a subprogram, package, or task unit.

Since the exception handler has the potential of being executed as a result of a transfer of control from any Ada statement in its current frame or from any Ada statement, operation, or elaboration of declarations in a nested frame, the number of possible paths to a handler cannot be determined by any method of static analysis. Also, the Ada language rules allow a large variety of Ada constructs to be coded within the handler, which may result in the recovery algorithm being as complicated or more complex than the abandoned algorithm. The third consideration regarding the replacement code or recovery code within an exception handler is that the handler actions are outside of (and often quite different from) the normal sequence of actions. These three characteristics of the Ada exception handler justify the ACE decision to compute the complexity of the exception handler separate from the complexity of the normal sequence of statements within the program module. The total module complexity is then represented by a composite number.

Graph 2 and Figure 2 illustrate a possible exceptional condition raised within the sequence of statements of a frame that contains a handler for the exception. Consistent with the intent of the Ada handler, when function AVERAGE raises CONSTRAINT_ERROR or NUMERIC_ERROR the sequence of statements within the handler are executed as a replacement for, or recovery from, the normal sequence of statements within function AVERAGE. The total complexity for function AVERAGE is a pair of values (3, 2). The first value represents the paths through function AVERAGE during normal execution; the second value represents paths in the exception handler(s).

When appropriate, a program module may contain several exception handlers. In this case, the second value of the composite complexity is the sum of the individual complexities of the handlers. Although the logic of the individual handlers may not be continuous, the second complexity value is a deliberate compromise designed to indicate the total complexity of all sequences of Ada statements outside the normal execution of the program module. The second value therefore approximates the amount of additional path testing necessary to verify proper operation of the program module.

4.4 Unconnected Flow Graph Segments

Unlike McCabe's traditional metric, which is based on a strongly connected graph, the ACE theory requires unconnected flow graph segments to accurately depict the flow of control. These occurrences of unconnected flow graph segments may be noted on Graphs 1 and 4.
Unconnected graph nodes may be caused by the following conditions:

- In any form of the Ada loop, execution of an unconditional exit statement results in a direct transfer out of the loop. Therefore, no arc should connect the exit node with any node inside the loop.

- An apparent "endless" loop may be legally coded in Ada. Therefore, there is no connecting arc to an Ada statement that follows the "end loop" node of any Ada basic loop that does not contain an explicit transfer statement.

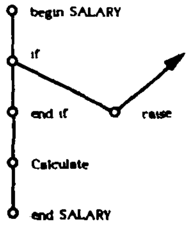
- By localizing the complexity analysis of an exception handler to those Ada statements within the handler it is inappropriate and often impossible to connect nodes associated with transfers of control outside the handler. Therefore, within an exception handler a node depicting an abort statement or raise statement may not be connected with any node in that program module.

4.5 Ada Raise Statement

Applying a similar analysis to the raise statement as was applied to the exception handler, the Ada raise statement should also represent a departure from the normal sequence of statements. The ACE proposes to represent the raise statement on the flow graph by a node with an exit path whose destination is undefined. The rationale for the unconnected exit arc is based on actions performed when the raise statement is reached during execution. When the raise statement is executed, execution of the normal sequence of statements within the current frame is abandoned in search of the exception handler named in the raise statement. This search may transfer program control flow to a handler at the end of the current frame or propagate it to an enclosing frame. Since the destination of the search transfer varies dynamically, it is not possible to statically determine an accurate connected flow graph. Procedure SALARY in Figure 3 and Graph 3 contains an illustration of a raise statement.

FIGURE 3

```
procedure SALARY (EMPLOYEE_DATA : in . . .;
                  RETURN_INFO   : out . . .) is
...
begin
  if EMPLOYEE_DATA.ID > 50_000 then
    raise EMPLOYEE_DATA_ERROR;
  end if;
  -- Calculate return info
end SALARY;
```



$$\begin{aligned} v\,(\text{SALARY}) &= e - n + 2 \\ &= 6 - 6 + 2 \\ &= 2 \end{aligned}$$

Graph 3

4.6 Ada Tasks

Process abstraction implemented by Ada tasks may involve a collection of sibling tasks, dependent tasks, tasks created from task types, and dynamically created tasks, all exercising a high degree of interaction through the rendezvous mechanism. The complexity of the dynamic characteristics of the Ada tasks cannot be evaluated by static analysis. However, an individual complexity value for each task body in the system can be determined by using the following ACE guidelines for analysis of the unique Ada statements that may appear in the implementation of the Ada task body.

4.6.1 Ada Abort Statement

Stopping execution of an Ada task without stopping execution of the entire program may be implemented by one task executing an abort statement. The abort statement may list a collection of task names, including the task object name that corresponds to the body containing the abort statement. If the task name of the task issuing the abort, or a master task, or an indirect master task of the task is included in the abort statement, then the "abort" node on the flow graph will be connected to the end node. When these task names are not included, the "abort" node is connected to the next sequence of actions. In either case, the evaluation of the abort statement does not contrbute to the complexity v.

4.6.2 Ada Select Statement

The select statement may be implemented in many forms, including the selective wait, the timed entry call, and the conditional entry call, employing such features as guard conditions, delay alternatives, the terminate alternative, and the else part. Although the evaluation of such features as guard conditions and the terminate alternative cause the decision mechanism within the select statement to become rather complex, the execution of this statement leads to a choice of just one of the select alternatives. The complexity evaluation of the select statement is analogous to the Ada case statement. That is, the selective wait, having x alternatives, will contribute x - 1 to the complexity of the task body module.

4.6.3 Task Entries and the Ada Accept Statement

When an entry call has been made and the corresponding accept statement has been reached, the rendezvous is executed by the called task and the calling task is suspended. Since the task entry synchronization is so closely dependent on the task owning the entry, the ACE considers the complexity of the sequence of statements in the accept statement to be an integral part of the task body complexity. As such, the complexity value associated with the accept statement is added to the task body complexity and the nodes and arcs that represent the paths within an accept statement are connected to the flow graph of the task body along the appropriate logical path. Figure 4 illustrates a select statement, three accept statements, and a basic loop coded within the Ada task body SCHEDULER.

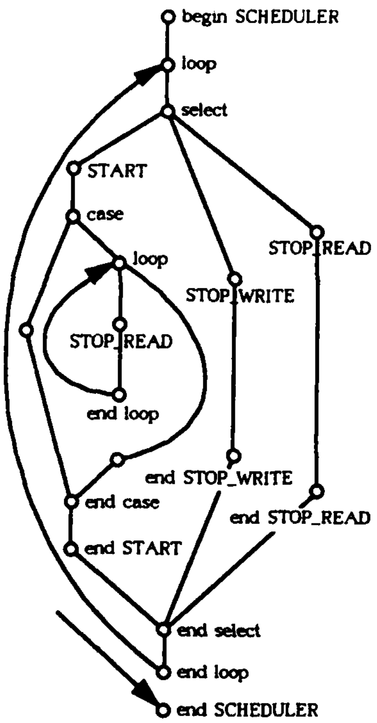
Several items can be noted in Graph 4. There are three branches corresponding to the three alternatives of the select statement. The paths within the three accept statements are connected to the flow graph of the task body. The basic loop contains no local means for completion. There is an incoming arc connected to the end node that indicates there is a means of unknown origin that will cause the task body to complete.

4.7 Additional Flow Graph Arcs

Certain compilable Ada constructs may be combined in such a way as to produce either unreachable program

FIGURE 4

```
WRITING  : Boolean := FALSE;
. . .
task body SCHEDULER is
  N_READERS : Task_Count := 0;
  . . .
begin
  loop
    select
      when not WRITING =>
      accept START ( REQUEST : in . . .) do
        case REQUEST is
         when READ  =>
           N_READERS := N_READERS + 1;
         when WRITE  =>
           for I in 1. .N_READERS loop
             accept STOP_READ;
           end loop;
           N_READERS := 0;
           WRITING := TRUE;
        end case;
      end START;
    or
      when WRITING =>
      accept STOP_WRITE do
        WRITING := FALSE;
      end STOP_WRITE;
    or
      when not WRITING =>
      accept STOP_READ do
        N_READERS := N_READERS - 1;
      end STOP_READ;
    end select;
  end loop;
end SCHEDULER;
```



$$v\ (\text{SCHEDULER}) = e - n + 2$$
$$= 23 - 19 + 2$$
$$= 6$$

Graph 4

segments or locally nonexecutable activities. An incoming arc that has an undefined source will be added to segments of the flow graph that are not connected in the forward direction. The additional arc will result in increasing the complexity value by one per arc. The increase is appropriate since these arcs will correspond to the test paths necessary to fully test all code paths within the program module. One example of an additional arc is illustrated at the bottom of Graph 4.

4.8 Ada Package

The Ada package specification is designed to encapsulate logically related entities and may include a subprogram, package, task, or generic declaration as a declarative item. If the package specification contains any of these declarative items, then the package body must contain the corresponding unit bodies in its declarative part. Those program unit bodies that are included in the declarative part of the package body should have the ACE complexity evaluation applied to each program unit that is considered a program module.

Consistent with the design of all Ada program unit bodies, the package body may contain an executable sequence of Ada statements with an optional exception handler before the end of the package body. If this optional sequence is present, the package body will then require a complex.ty evaluation incorporating ACE guidelines for those Ada statements used within the executable sequence of that package body.

4.9 Ada Generic Units

The Ada generic unit is a template in the form of a generic subprogram or generic package. Although the body of a generic unit does not represent an executable Ada program unit, it is necessary to include it as an analyzable program module. The reason is that the source code within the generic body represents the only accessible reference for analysis. Each time an executable subprogram or package is obtained from a generic unit, the compiler performs the substitution of the actual parameters for the generic formal parameters and the resulting nongeneric unit becomes part of the program. Since the nongeneric bodies are very similar in logic structure to that of their corresponding generic unit, the complexity of the instantiated program unit will be considered identical. A complexity value is only valid when the program module contributes to the complexity of the program. Therefore, the complexity measure computed for the generic body is to be applied only to each instantiation of the generic unit.

5. FUTURE RESEARCH AREAS

The ACE metric recommendations that are provided in this article for extending McCabe's cyclomatic complexity metric have been evaluated and tested on a limited amount of Ada software. Extensive empirical evaluation of the ACE will be necessary to "fine-tune" the theory and maximize its practicality.

There is support for the concept of expected maintenance difficulties and error-prone modules correlated to high complexity design and program modules. Consequently, interest has been expressed in determining a value that represents a reasonable upper limit for the complexity of an Ada software module. Empirical studies [5] have supported the number 10 as a practical upper-limit guideline for the application of McCabe's metric to traditional high-order languages. In addition, the use of a composite complexity value associated with program modules containing exception handlers may be modified to indicate the number of exception handlers within the program module. Guidelines for the maximum number of handlers per module and for determining a reasonable upper limit for handler complexity are also under consideration.

The manual evaluation of the complexity of an Ada program module according to the ACE metric can be tedious and may be inaccurate because of the variety of Ada constructs and features. Producing reliable calculations of the complexity measure will require an automated Ada complexity analysis tool [2]. Such a software tool would provide useful support to an Ada program development environment.

6. CONCLUSIONS

The criteria applied to evaluate the usefulness of a proposed software metric include a straightforward and intuitively appealing interpretation and its practical applicability to multiple phases of the software life cycle. The ACE metric responds to both criteria by using either a single or a composite value to quantify the logical complexity of a program module. The complexity value has a direct correspondence to the number of basis paths [3] through that module. The ACE metric supports evaluation and provides assistance through the design, implementation, test, and maintenance phases.

The ACE metric provides for managers and developers the visibility and quantifiability of Ada module complexity. Further research and information derived via empirical case studies will be needed to refine and validate all aspects of the ACE metric. The application of an automated ACE complexity analyzer [2], together with other Ada environment tools, would assist software development by enhancing the reliability and maintainability of Ada software systems.

7. ACKNOWLEDGMENTS

My sincere gratitude is expressed for the technical, editorial, and graphical assistance given by Charles A. Finnell. I am also appreciative of the technical assistance and contributions given by Joseph P. Salvador, Grace Baratta-Perez, Donald R. Clarson, and Thomas J. Walsh of Teledyne Brown Engineering. Thomas J. McCabe, President of McCabe and Associates, has reviewed the ACE theory and offered comments that have been addressed in this article. The research effort was partially funded by the Software Quality Assurance Group of the U.S. Army Armament, Munitions and Chemical Command.

8. REFERENCES

[1] H. J. Tauson-Conte, J. P. Salvador, C. A. Finnell, G. Baratta-Perez, D. R. Clarson (authors), "Extending McCabe's Cyclomatic Complexity Metric for Analysis of Ada Software," Technical Report MC87-McCabe II-0003 prepared for Director, Product Assurance, U.S. Army AMCCOM, March 1987.

[2] H. J. Tauson-Conte, G. Baratta-Perez, C. A. Finnell, D. R. Clarson (authors), "Modified A-Level Software Design Specification for the Ada Complexity Analysis Tool which Automates the ACE Metric," Technical Report MC87-McCabe II-0005 prepared for Director, Product Assurance, U.S. Army AMCCOM, April 1987.

[3] McCabe, T. J., "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976.

[4] McCabe, T. J. (editor), "Structured Testing," IEEE Computer Society Press, IEEE Catalog No. EH0200-6, 1982.

[5] Walsh, T. J., "A Software Reliability Study Using a Complexity Measure," Proceedings of the 1979 National Computer Conference, AFIPS Press, 1979.

[6] National Bureau of Standards, "Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric," 1982.

[7] ANSI/MIL-STD-1815A: Ada Programming Language, U.S. Department of Defense, February 1983.

Biographical Sketch

Holly Tauson-Conte is a principal systems analyst and has been involved in Ada technology for over 3 years at Teledyne Brown Engineering. She was an assistant professor from 1981 until 1984 in the Computer Science Department at Monmouth College. Prior to that, she was a computer scientist for 5 years at Bell Telephone Laboratories. Tauson-Conte's areas of interest cover software metrics, Ada programming support environments, software engineering, and parallel processing. Tauson-Conte received her B.A. degree from Montclair State College in mathematics and her M.S. degree from Monmouth College in computer science. She is a member of the ACM, the SIGAda, and IEEE.

AN INVESTIGATION INTO THE COMPATIBILITY OF ADA AND FORMAL VERIFICATION TECHNOLOGY

DR. DAVID PRESTON
IIT Research Intitute
4550 Forbes Blvd.
Lanham, MD 20706

MR. KARL NYBERG
GREBYN, INC.
P.O. Box 1144
Vienna, VA 22180

DR. ROBERT MATHIS
CONSULTANT
9712 Ceralene Dr.
Fairfax, VA 22032

ABSTRACT

The goal of this study is to investigate approaches to 'high-assurance' software written in the Ada programming language. 'High-assurance' software includes the software in systems defined to be 'secure' by the Department of Defense Trusted Computer System Evaluation Criteria (TCSEC) as well as other software with very high reliability or security requirements. The primary approach to high-assurance software considered here is formal code verification. This report investigates Ada constructs relative to code verification, the technologies necessary to support code verification, and ongoing efforts directly related to verification of Ada code. Since Ada was not developed to be a verifiable language, there are some constructs that will defy formal verification; these challenges do not seem to be overwhelming and could presumably be controlled by restrictions on the use of the language. Tasking and exception handling are the two greatest challenges that the language constructs provide for verification, with tasking being the greater challenge.

INTRODUCTION

This paper addresses several issues. The abstract goal is to investigate methods that would lead to a 'high assurance' that software written in the Ada language would perform as intended. To make this goal concrete required a preliminary understanding of 'high assurance.' The Department of Defense Trusted Computer Systems Evaluation Criteria defines secure computer systems as those that satisfy six requirements:

1. There must be an explicit and well-defined security policy enforced by the system.
2. Access control labels must be associated with objects.
3. Individual subjects must be identified.
4. Audit information must be selectively kept and protected so that actions affecting security can be traced to the responsible party.
5. The computer system must contain hardware/software mechanisms that can be independently evaluated to provide sufficient assurance that the system enforces requirements 1 through 4 above.
6. The trusted mechanisms that enforce these basic requirements must be continuously protected against tampering and/or unauthorized changes.

The software in these systems is frequently referenced as 'secure' or 'trusted' software. The conclusions of this paper are directly applicable to 'secure' and 'trusted' software and are also applicable to a much broader collection of software. This broader collection includes software with very high reliability or security requirements, and software which must function as intended or there will be threat to human life or national security. Throughout this report, such software is referred to as 'high-assurance.'

The initial approach of the research on which this paper is based was a review of each Ada construct as defined by the Ada Language Reference Manual (LRM). This review was centered on the impact of each construct on formal verification. The assessment was based on the feasibility to develop a verification axiom, or proof rule, for each construct in isolation. Those constructs with the greatest impact on verification are reported here.

In addition to the axioms, other support technologies are required for code verification. These include a formal definition and a specification language. Specification languages are not only necessary for formal code verification, but can also be used with other techniques, both formal and less formal. An understanding of runtime issues is necessary to understand the limitations of code verification relative to how the software will function during execution. The status of each of these issues relative to Ada execution is presented.

Some of the conclusions of this study go beyond the initial study plan. While investigating the principle questions addressed by the study, secondary observations were made and are included.

ADA CONSTRUCTS THAT AFFECT VERIFICATION

Ada is generally viewed as a rich language. The richness of the language is perceived to be detrimental to formal verification. This section highlights the constructs most challenging to verification, identifies a few problems considered to be unresolvable and outlines restrictions on Ada programming style that would be necessary if the code were to be formally verified. The conclusion of this section outlines the impact that using Ada would have on secure systems.

Different perspectives are used in different subsections of this section. The subsections on challenging constructs, unresolvable problems, and necessary restrictions in coding style assume that the intent is to verify code using axiomatic verification techniques on the code. The concluding subsection on the impact of the use of Ada for secure systems takes a pragmatic view, assessing the impact of Ada on current practice.

Most Challenging Constructs

Tasks, Ada's implementation of concurrency, are the most challenging obstacle to applying formal verification technology to the Ada language. By allowing only restricted use of tasking, it appears that concurrency in Ada can be made amenable to application of formal verification technology; however, it remains to be seen whether what remains has any semblance to what would be called Ada, and whether it would have any usefulness for the objectives for which it was designed into the language. Use of the techniques employed in Communicating Sequential Processors (CSP) [Barringer] appears to be a promising possibility while other research indicates that restriction of communication to only buffers (a la Gypsy) [Young80], to only scalars [Odyssey85], or to only those entry points in which pre-condition and postcondition assertions have been specified [Tripathi] would alleviate many of the inherent difficulties in applying formal verification technology to the Ada concurrency problems.

It must be noted that those restrictions to communicating between tasks reflect the state of the art rather than assess feasibility. Although no one has published proof rules for passing aggregate types, for example arrays or records, development of such proof rules seems quite feasible.

Several researchers [Odyssey85, Pneuli] have recommended that access pointers to tasks not be allowed. The intent of this restriction is to disallow dynamic creation of tasks. In the absence of dynamic creation of tasks, proof rules can be obtained for tasking. However, it is unlikely that this limitation will be readily accepted, particularly in the systems programming arena.

The approach used [Owicki and Gries] to verify Communicating Sequential Processes, Hoare's language framework for concurrent programming, is readily adapted to verification of Ada tasks. This approach consists of two distinct steps: internal verification and external verification. Internal verification consists of proving that the task is an isolated, sequential program. External verification consists of proving that, with the exception of entries, tasks do not affect any subprograms, tasks, or variables declared outside of the task being verified. External verification also requires proof that the task in question is not affected by any subprograms, tasks, or variables declared outside of the task. Again, entries are the exception to this rule. External verification is performed in two states:

a) I-O assertions on entries are made and shared variables are restricted.

b) A proof against deadlocks and starvation is made.

Deadlock and starvation avoidance proofs are prevalent throughout parallel processing literature.

The verification of tasks also assumes the following:

a) All processes terminate normally.

b) Subprogram calls have no side effects.

c) Assignments have no side effects.

d) Tasks may not be aliassed.

Exceptions are the other principle obstacle to verification. The major difficulty with exceptions [Tripathi] in the Ada language from the point of view of verification is the dynamic manner in which exceptions are propagated, and the resulting complexity that derives from attempting analysis during symbolic execution of programs in the verification step. This complexity is furthered by the fact that exceptions are propagated "as is," which could cause an unhandled exception to propagate from several levels down to a routine that has no understanding of the meaning of the exception. For example, a stack package with a private implementation that raises INDEX_ERROR in the environment of the calling procedure would be totally unexpected and either unhandled or mishandled.

Through adequate containment of the exceptions, the complexity should be reduced. However, the interaction of exceptions and other constructs moves this issue well beyond the problem of bookkeeping. For example, if an exception is raised during execution of a routine with IN OUT parameters, it is not clear if those variables will have been updated prior to transfer of control to the exception handler.

Unresolvable Problems

Programming languages not developed for verification inevitably contain constructs that are non-verifiable. Some of these can be controlled through restrictions on programming practices and are discussed in the next section. For two common programming constructs there is no current solution. Although these constructs are not unique to Ada, they do exist in Ada.

Verification of statements including real numbers, and operations on real numbers, is beyond the state of the art. This is due to the lack of accuracy. In the statement

```
J:= (1.0/3.0) + (1.0/3.0) + (1.0/3.0)
```

J would, mathematically, be set to 1.0. However, not only is it uncertain if J will equal 1, it is not known how close to 1 J will be. The effect of this on subsequent statements involving J is unpredictable. As in other languages using real numbers, they cannot be used if the software is to be verified.

Another area that is not verifiable is the process of setting timing constraints. If a section of code must be executed within a specified time, there is no way to verify that the constraint will be met.

Restrictions to be Enforced

The recommended coding restrictions of note involve aliasing, aliasing of access types, using shared variables by tasks, and side effects of functions.

Verifying a specific subprogram call requires verifying certain conditions about the parameters involved in the call. These parameters fall into one of two categories: input parameters or output parameters. Input parameters are used only for passing values to the subprogram; output parameters may have their values altered by the subprogram. The conditions that must be verified for each are as follows. No variable, either input or output, may appear in either the precondition or postcondition. No variable that appears in the output parameter list may appear more than once in that list, and no output parameter may appear as an input parameter. The former condition results in updating multiple variables when only one is intended to be updated. For example, if two subprogram formal parameters, A and B, are both passed variable X through a subprogram call, the result of the statements

```
A:=0
B:=1
```

leaves variable X with the value 1 and no variable from the call with value 0. The postcondition after these two statements would assume the existence of two distinct parameters, one with value 1 the other with value 0. If an output parameter appears as an input parameter, the time at which the input parameter is evaluated becomes critical. If the output variable is updated prior to the evaluation of the input parameter, the value of the input parameter may differ from the value recorded if the output parameter is not updated prior to evaluation of the input parameter. A single actual parameter used for more than one formal parameter is known as "aliasing."

The association of parameters at subprogram call points would be the ideal location to exclude aliasing [Good80, Odyssey85]. Although there might be a loss of efficiency, the fact that aliasing is unnecessary and complicates application of formal verification technology [Young81] would seem to be sufficient reason for its elimination.

The major concern in the use of access types is the possibility of aliasing. One possible solution to the aliasing problem with access types, presented in [Tripathi], is to define a new operator for access types that performs component copying, rather than pointer duplication. This solution is appealing with the advent of the evaluation of the Ada language, due in the latter part of the 1980s, when changes and updates based on several years of working experience with the language will be incorporated into the language. However, restrictions on parameter passing [Odyssey85, Young81] would appear to provide the same benefit with fewer changes.

If a function performs input or output or accesses *non-local* variables, it is said to cause "side effects." If function subprograms are truly functional they will not include side effects. If Ada functions are restricted to exclude side effects, they can be verified similarly to Gypsy function subprograms, in which these restrictions are enforced by the language.

Shared variables are the major construct in tasking that will have to be restricted (although perhaps simulated through use of other constructs using synchronization) in order to apply formal verification technology to Ada. On this matter, there is no disagreement among the researchers [Cohen, Good80, Odyssey85, Tripathi].

Many of these recommended restrictions are consistent with what are considered good programming practices. The exception is use of shared variables by tasks; forcing tasks to communicate by other means will restrict the utility of tasking. If the code is to be verified, however, the restriction may be necessary.

FORMAL VERIFICATION IN ADA

Formal verification is the highest technology approach to increasing assurance in the correct functioning of computer software. Other approaches, such as testing, configuration management, or development methodology, have benefits but verification alone can make a quantum leap in the level of assurance. This has resulted in the unfortunate position that verification is an all-or-nothing proposition for software development requiring very high levels of assurance. This mentality has only slowed the application of formal verification technology.

The formal verification process consists of preparing, prior to the development of software, the formal specification of a model of the intended behavior of the software. Some effort may be placed (as described previously) in the analysis of the specifications to ascertain their completeness and internal consistency. Then, following the software development methodology, designs and implementations at the various levels of the software are completed, and formal correspondence with the specification is performed, resulting in proofs of correctness of the implementation with respect to the specification. The formal proof of correctness consists of showing that the two separate, hopefully somewhat orthogonal, descriptions have a proper correspondence.

Assessing the state of the art of formal verification technology relative to the Ada language requires perspective. To date, the largest code verified system in operation is 4,211 lines of code. Given that languages that are designed and developed to be verifiable provide challenges to the development of large, complex systems, it would be naive to expect that Ada would be easily verifiable.

One of the earmarks of formal verification technology is its formality. This is an area that seems to have had varied amounts of support during the design of Ada. In the early requirements documents for the Ada language, verification was mentioned as a desirable goal, but the language contains many constructs that prevent this goal. In order to do formal proofs of consistency between the specification and the implementation, a formal description of the language semantics is also necessary. Some effort has been done by the EEC in this area, but it has not sufficiently matured to a stage where it can be utilized in formal verification. The only viable specification language for Ada, ANNA, has been geared more toward the utilization of runtime assertion checks, not formal verification, and has largely ignored the aspects of parallelism. At least one known effort is involved in extending ANNA to overcome these deficiencies.

Once the remaining theoretical obstacles have been overcome, it will be necessary to develop automated support tools for the specification and verification process.

ADA-SPECIFIC SUPPORT TECHNOLOGIES

Formal code verification requires several key components. The implementation language must have a formal definition or semantics so that the exact meaning of each language construct and sub-construct is clear and unambiguous. There must be a specification language. Since the proof establishes the consistency between the specification and the code, the specification must be stated in a formal language. Although not required for code verification, runtime issues are very important. Code, even if proven correct, will not function as expected if the runtime environment does not execute in a manner that is consistent with the assumptions of the proof.

These issues -- formal definitions and semantics, specification languages, and runtime issues -- are discussed in this section.

Formal Definition and Formal Semantics

The most ambitious attempt at a formal definition of the Ada language is being undertaken by the Dansk Datamatik Center and its member companies. This definition is intended to give meaning to each Ada language construct by providing meaning to each sub-construct. This definition is to be a readable, unambiguous definition that will be implementation dependent. The approach was to develop a static semantics and then to develop the dynamic semantics. The dynamic semantics will have embedded in it the sequential constructs, as these may be executing in parallel, and the input-output portions of the language.

The semantics are provided by use of axioms which are given as abstract data types and an algebra, or model, for combining the constructs. In addition to being a basis for formal proofs, this definition is meant to be a standard reference or specification for implementors of the language.

This effort has produced a very large volume for the formal definition. Although the developers have built into the definition mechanisms to establish the completeness and consistency of the definition, these two concerns -- consistency and completeness -- are still major.

SofTech has been working on an effort to define the problems and potential solutions to the development of an axiomatic semantic definition of the Ada language. The difference between an axiomatic semantic description of Ada and the definition of Ada given by MIL-STD-1815A is that the semantic description defines the behavior and interrelationships of the individual language constructs in such a way as to be used as the basis of a proof. The existence of a semantic definition of a language is necessary if a comprehensive verification technology is to be developed for that language. Any aspects of a language that are not rigidly, semantically defined are subject to varying interpretations by different compilers. Some of the Ada constructs that pose difficulties in verification have been left out of the semantic description. A list of the excluded constructs are address clauses, unchecked conversions, variables shared among tasks and subprogram calls that generate aliases.

To a large extent, most elements of a semantic description are handled at compilation time and need not be dealt with during verification time. It is important to realize that the actual verification environment is based on the semantic definition of Ada rather than the actual language, and constructs that are not included in the semantic definition invalidate the verification process. The SofTech study concerns itself only with those constructs that are not dealt with at compilation time.

Specification Languages

Specification languages are necessary for code verification and can also be used for other proof-related purposes. Analysis of the specifications prior to proving consistency between the code and the specification, can only be formally done with a mathematically-based specification language. Also, runtime analysis is facilitated by use of a specification language to state the assertions that are to be checked at runtime. Some Ada-specific work on specification language tools is being done at Stanford University.

At the present time, software system specifications are done in the English language. While using English as a specification language has the advantage of providing easily readable, easily composed specifications there are some problems inherent with the use of English. The English language often contains inconsistencies and ambiguities which inhibit exact interpretations of the specifications. The translation required from specification language to coding is so broad due to the vast difference in media as to create transitional errors in all but the most detailed, trivial or exhaustively used system.

The principle specification language for Ada was developed at Stanford University. ANNA (ANNotated Ada) is an annotation language for all constructs of Ada except tasking. The language is designed to support various theories of formally specifying and verifying programs. One area of current research is the use of parallel processors to provide concurrent checking of specifications.

Since the ANNA semantics closely parallel those of Ada, its use in secure systems development would allow the system designers and implementors to use the same underlying language semantics for communication of the intended behavior of their specifications and programs. However, since the language appears to have been targeted to the runtime validation of program execution rather than pre-execution proofs of correctness, its applicability in secure systems development would be limited until a supporting infrastructure, both in terms of theoretical aspects of the language and in terms of automated tools, can be developed.

The use of Ada as its own specification language would enable the specification to be read and interpreted by a compiler-like consistency checker which is able to enforce internal consistency within the semantics. Taken to a higher level, the consistency checker may be used to check the consistency between different levels of specification. In this manner the integrity of the initial specification may be checked, level by level, down to the actual code. The disadvantage of using Ada as a specification language is the limitation of Ada's expressibility relative to specifications.

Runtime Issues

Verification of a program, in any language, takes place during a "proof time" which occurs before the program is executed. Situations that are difficult to predict at proof time are generally either discounted or disallowed by verification techniques. The result of this is that one of two things happens:

either an issue is ignored or discounted in some superfluous way, or a great deal of effort is spent attempting to suppress possible occurrences of the problem.

Ada deals with runtime difficulties through the use of exceptions. Much work has gone into exception handling during verification. One of the more detailed investigations into runtime issues relative to verification is McHugh's work at the University of Texas at Austin on the Gypsy language [McHugh]. Gypsy's exceptions are similar to Ada; this enables us to apply runtime techniques developed in Gypsy to Ada.

McHugh handles exceptions in two manners, one of which is that exceptions that are considered domain related. These exceptions are discounted with regards to verification. An illustration of this is as follows: a verified satellite communications system would fail if the satellite were disabled. The effect of this decision is to localize the responsibility of the verification to not include errors that emit from outside the program. Should the satellite be verified in addition to the software, then a failure would indicate a fault in the verification process.

Exceptions that are not external in origin are handled differently. These exceptions are, in effect, eliminated from the program to be verified. Exceptions of this type are indirectly optimized during the verification process before runtime. This is performed by the creation of optimization conditions that are related to possible exceptions. Optimization conditions must be sufficiently well defined to show that the corresponding optimization condition must occur before the exception may be raised. Given this, it is easily proven that, if an optimization code can be proven to never occur, the exception will never be raised. It is easily concluded that an exception which is never raised cannot compromise the verification of a program or module of a program. We may now state that, if an optimization condition is offered as a precondition of a module of Ada code then the code may be considered verified with respect to the exception that corresponds to the optimization condition.

In short, the nature of exceptions makes them difficult to verify in a robust manner. The state of the art is little more than the statement that "if an exception never occurs it creates no problems in verification."

Runtime Assertion Checks

Another area in which specifications may be applied is that of runtime assertion checks. This subsection describes the various types of such checks, their applicability and utility, and the status of the technology as applied particularly to Ada.

Runtime assertion checks can increase the assurance in the correct functioning of a program in a number of ways. First, the additional effort expended in the development of such assertions, whether they be informal or formal, increases the level of the programmer's understanding of the program. Second, the preparation of assertions can provide a gentle introduction to the application of formal verification technology, by allowing the programmer to get a small amount of exposure to part of the verification process without having to make the total investment in learning the process at once. Finally, and the major reason for their use, is that the runtime checks can be used in instrumented versions of the executable programs to check the programmer's understanding of the program against its actual execution.

Runtime assertion checks can be included in a program in various forms. The first, and most obvious, form of assertion is simple inclusion of code in the programming language itself. This code may be instrumented in such a way as to be turned on or off at runtime, although recompilation of the source code may be required. The level of overhead associated with such checking increases from the lowest, in which the runtime assertions are not included at compile time, followed by instrumentation with checks turned off, runtime assertions compiled in, to instrumentation with checks turned on. Another possible form of inclusion of runtime assertions is through the use of a formal assertion mechanism. These may be processed by a preprocessor, as in the case of the C language assert construct, and converted into corresponding source code, or parsed with the program text, as in ANNA runtime specifications, and expanded during the code generation phase of compilation.

One benefit of the use of runtime assertion checks is that the technology is so similar to compiler technology that it can be applied without additional technology development. Its application is also at a sufficiently low level to allow its use by programmers at various ability levels (depending upon the level of formality of the specification language). Another benefit is that the technology can be easily integrated into the traditional software development methodology without having to make large investments in retraining, changes in practice, or additional hardware.

The use of runtime assertion checks is not without drawbacks, however. The most obvious one is the overhead penalty in execution time while running programs instrumented with such runtime assertions. Another drawback is that the application of formal verification technology may obviate such runtime checks. With a formal verification methodology, it may be possible to logically prove that the assertion holds at the point in the program's execution, and the resulting runtime check can be omitted, thus reducing the program's runtime overhead and increasing its performance.

Runtime assertion checking is a technology which can be, and currently is being, applied to increase the assurance in the correct execution of software written in Ada. Research at Stanford University has resulted in ANNA, a specification language for Ada, specifically designed for use in preparing runtime assertion checks. Automated tools for supporting the software development process using such checks have been developed, and preliminary results have been obtained on a number of research and development projects.

ANNA was designed primarily for use with the sequential aspects of the Ada language. Efforts are underway to extend ANNA and combine it with other languages to use it for the parallel aspects as well. Additional research is being targeted at providing a mechanism for concurrent execution of the resulting runtime assertion checks (on multi-processor hardware) in order to exploit some of the benefits of the parallel execution and reduce the apparent runtime overhead penalty associated with the checks.

The use of ANNA is very CPU intensive, not only in the execution time of the resulting software, but also in the execution time of the automated tools. The lack of speed in these tools is due in part to their use of somewhat dated compiler technology. This drawback might prevent its application in environments that are unable to provide a sufficient hardware base for development environments. The benefit of the choice of Ada has allowed transition among available hardware configurations which provide Ada software development environments.

CONCLUSIONS

This study had several objectives. The abstract goal was to investigate methods that would lead to the highest assurance that software written in the Ada language would perform as intended. This led to the examination of elements related to the formal verification of Ada software, to the examination of formal methods applied at levels other than code verification, and to the examination of less-formal methods.

Relative to code verification, the continuing examination of Ada constructs reveals two findings. Since Ada was not developed to be a verifiable language, there are some constructs that will defy formal verification; these challenges do not seem to be overwhelming and could presumably be controlled by restrictions to the use of the language. Tasking and exception handling are the two greatest challenges that the language constructs provide for verification. Of these, tasking is the far greater challenge.

Code verification requires both a formal definition and a specification language. The formal definition being developed by DDC will need to be verified, validated or certified by someone outside of the developing group. This is a major issue. Also, the structure and syntax of the definition will limit its utility.

ANNA as a specification language has limitations which are being addressed by

Odyssey Research, and alternative forms for specifications are being investigated by Computational Logic.

As these various elements of formal verification with the Ada language progress, it will remain to apply resulting technology in order to gain experience with it and to evaluate the feasibility for development of large scale projects. To date, applications of verification technology have been performed by small groups of people on small tasks, with rather limited results in terms of both costs and quantity of software. This situation will be no different in application of formal verification technology to the Ada language. The community of individuals trained in the use and application of formal verification techniques is small, and the intersection of those individuals with the limited pool of talent proficient in the Ada language continues to reduce the available labor.

Existing verification projects have been small in size, because that has been the only manner in which to maintain control of the complexity of the project, and to be able to support the project with automated support tools. Advances in hardware technology will improve the utility of support tools, but cannot solve the personnel problem.

Beyond code verification, formal methods are being investigated with respect to Ada. The two areas of research are the application of formal methods to specification analysis and to runtime assertions. In the area of specification analysis, the focus is on finding and analyzing inconsistencies in the specifications. Although this will not provide the assurances of code verification, it seems that an emphasis on this work will prove to support code verification in the long run, and will be useful in its own right. Successful verification projects depend on consistent specifications.

Although runtime assertion checking seems redundant for verified software, this avenue seems particularly worthy of research for distributed systems. This avenue may well help in the understanding of concurrency.

Assessing the adequacy of the state of the art of formal verification technology relative to the Ada language requires perspective. To date, the largest code verified system in operation is 4,211 lines of code. Given that languages that are designed and developed to be verifiable provide challenges to the development of large, complex systems, it would be naive to expect that Ada would be easily verifiable.

REFERENCES

Ada Programming Language, ANSI-MIL-STD-1815A, Department of Defense, 22 January 1983.

Barringer, H., Mearns, I., "Axioms and Proof Rules for Ada Tasks," IEEE Proceedings, Volume 29(E), Number 2, pp. 38-48, March 1982.

Cohen, Norman H., "Ada Axiomatic Semantics: Problems and Solutions," SofTech, Inc., One Sentry Parkway, Suite 6000, Blue Bell, PA 19422-2310, May 1986.

DiVito, Ben, "A Verifiable Subset of Ada," TRW, unpublished manuscript.

"The Draft Formal Definition of ANSI-MIL-STD 1815A Ada," EEC Multi-annual Programme, Project No. 782, Annex 1, Version 14-12-1984, Dansk Datamatik Center, Lundtoftevej 1C, DK-2800 Lyngby, Denmark.

Ernst, G.W., and Hookway, R.J., "Specification and Verification of Generic Program Units in Ada," Department of Computer Engineering and Science, Case Institute of Technology, Case Western University, Cleveland, Ohio.

Gerth, R. "A Sound and Complete Hoare Axiomatization of the Ada Rendezvous," Proceedings of the 9th International Coloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science 140, Springer Verlag, pp. 252-264, 1982.

Gerth, R. and deRoever, W.P., "A Proof System for Concurrent Ada Programs," RUU-CS-83-2, Rijksuniversiteit Ultrecht, January 1983.

Good, Donald I. et al., "Report on the Language Gypsy," Version 2.0 The University of Texas at Austin, ICSCA-CMP-10, September 1978.

Good, Donald I., Cohen, Richard M., and Keeton-Williams, James, "Principles of Proving Concurrent Programs in Gypsy," Institute for Computing Science and Computer Applications, The University of Texas at Austin, Austin, TX 78712, January 1979.

Good, Donald I. et al., "An Evaluation of the Verifiability of Ada," September 1980.

Goodenough, John B., "Exception Handling: Issues and a Proposed Notation," Communications of the ACM, 18(12):683-696, December 1975.

Hill, A.D., "Asphodel - An Ada Compatible Specification and Design Language," (unpublished manuscript), Central Electricity Generating Board, Computing and Information Systems Department, Laud House, 20 Newgate Street, London EC1A 7AX, U.K.

Ichbiah, J.D., Barnes, J.G.P., Heliard, J.C., Krieg-Brueckner, B., Roubine, O., Wichmann, B.A., "Preliminary Ada Reference Manual" and "Rationale for the Design of the Ada Programming Language," ACM SIGPLAN Notices, 14(6), June 1979.

Institute for Defense Analyses, "Proceedings of the First IDA Workshop on Formal Specification and Verification of Ada," HQ85-29920-1, Institute for Defense Analyses, 1801 N. Beauregard St., Alexandria, VA 22311, May, 1985.

Kemmerer, Richard A., Verification Assessment Study, Final Report, National Computer Security Center, C3-CR01-86, Ft. George G. Meade, MD 20755-6700, March 27, 1986.

Formal Definition of the Ada Programming Language, Institute National de Recherche en Informatique et en Automatique, November, 1980.

Liskov, Barbara H., and Snyder, "Alan, Exception Handling in CLU," IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, pp. 546-558, November 1979.

Luckham, David C. and Suzuki, Norihisa, "Verification of Array, Record, and Pointer Operations in Pascal," ACM Transactions on Programming Languages and Systems, pp. 226-244, October 1979.

Luckham, David C. and Polak, Wolfgang, "Ada Exception Handling: An Axiomatic Approach," ACM Transactions on Programming Languages and Systems, 2(2):225-233, April 1980.

Luckham, David C., von Henke, Friendrich W., Krieg-Brueckner, Bernd, Owe, Olaf, "ANNA -A Language for Annotating Ada Programs, Preliminary Reference Manual," Technical Report No. 84-261, Program Analysis and Verification Group, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, July 1984.

McGettrick, Andrew D., Program Verification Using Ada, Cambridge Computer Science Texts - 13, Cambridge University Press, 1983.

McHugh, John, "Towards Efficient Code from Verified Programs," Technical Report ICSCA-40, Institute for Computing Science, University of Texas at Austin, March 1984.

Mills, Harlan, "Human Verification in Ada," Presented at the Third IDA Workshop on Ada Specification and Verification, Research Triangle Park, NC. May 14-16, 1986.

Nyberg, Karl A., Hook, Audrey A., Kramer, Jack F. "The Status of Verification Technology for the Ada Language," Institute for Defense Analyses Paper #P-1859, IDA, 1801 N. Beuregard St., Alexandria, VA 22311, July 1985.

Odyssey Research Associates, Inc., "A Verifiable Subset of Ada," (Revised Preliminary Report), Odyssey Research Associates, Inc., 301A Harpis B Dates Drive, Ithaca, NY 14850, September 14, 1984.

Odyssey Research Associates, Inc., "Toward Ada Verification," Preliminary Report (Revised Preliminary Report), Odyssey Research Associates, Inc., 301A Harpis B Dates Drive, Ithaca, NY 14850-1313, March 25, 1985.

Owicki, S.S., and Gries, D. "Verifying Properties of Parallel Programs: An Axiomatic Approach," Communications of the ACM, Vol. 19, No. 5, (May 1976), 279-285.

Pneuli, A., and deRoever, W.P., "Rendezvous with Ada - A Proof

Theoretical View," Proceedings of the AdaTEC Conference on Ada, Arlington, VA, pp 129-137, October 1982.

Tripathi, Anand R., Young, William D., Good, Donald I., "A Preliminary Evaluation of Verifiability in Ada," Procedings of the ACM National Conference, Nashville, TN, October 1980.

Young, William D., Good, Donald I., "Generics and Verification in Ada," Proceedings of the ACM Symposium on the Ada Language, Boston, MA, pp. 123-127, 9-11 December 1980.

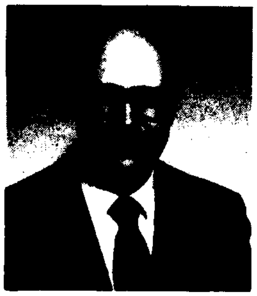
Young, William D., Good, Donald I., "Steelman and the Verifiability of (Preliminary) Ada," ACM SIGPLAN Notices, 16(2):113-119, February 1981.

BIOGRAPHICAL SKETCHES

Dr. David Preston is a Senior Software Engineer with IIT Research Institute and an adjunct faculty member of the Computer Science Department of the University of Maryland. His specific research interests are the use of Ada for secure systems, real-time application issues, and runtime environment evaluation techniques and criteria. He is a member of IEEE, the IEEE Computer Society, and the IEEE Technical Committee on Software Engineering. He holds a B. S. in Earth and Space Science from Clarion State College, an M. S. in Mathematics from Ohio University, and a Ph. D. in Mathematics Education and an M. S. in Computer Science from the University of Maryland.

Mr. Karl Nyberg is founder and president of Grebyn Corporation. His areas of research include Ada, UNIX, formal methods and computer security. He is a member of the ACM and IEEE. Mr. Nyberg received his B. S. in Electrical Engineering and M. S. in Computer Science from the Massachusetts Institute of Technology.

Dr. Robert F. Mathis served as the Director of both the Ada and STARS programs in the Office of the Secretary of Defense until 1985 when he left DoD for private practice as a consultant on software technology for systems development and support. Before joining DoD, he was on the computer science faculty at Ohio State and Old Dominion Universities. He holds a B. Sc. in Mathematics, an M. Sc. in Statistics, and a Ph. D. in Numerical Analysis, each from Ohio State University.



Ada SOFTWARE METRICS

LTC Richard P. Delaney, U. S. Army,
AWIS/CCS Project Management Office,
Fort Belvoir, Virginia
and
Mr. Lawrence F. Summerill,
TRW, Defense Systems Group
Fairfax, Virginia

ABSTRACT

Metrics are the quantification of environmental and performance factors to measure the effectiveness of activities in the areas of resources, schedule, quality, and risk. Metrics provide both a prospective and retrospective measure of accomplishment. Retrospective data provides a baseline for the next project. Prospective data support forecasting, planning, and control of on-going activities. The latter is obviously preferrable.

This paper summarizes the types of metrics developed during the foundation phase of the Army WWMCCS Information System (AWIS), and the methodology applied to achieve a selected subset of these metrics. Current plans are to continue to tune and refine these metrics during full scale development, which starts early in 1988 and is expected to last for five years.

INTRODUCTION

Project Managers strive to be in the forefront of technology. However, along with the distinction of being a technological first come problems that do not surface in other projects that follow. Both of these statements are true of the Army WWMCCS Information System (AWIS) Software Development project.

The AWIS Software Development is being done in Ada and is the first large scale non-embedded Ada development within the Army. AWIS is the Army portion of the Joint modernization of the World Wide Military Command and Control System (WWMCCS). WIS (WWMCSS Information System) is also being developed in Ada, but in many respects AWIS software development is ahead of the WIS software developments. AWIS has started the designing and coding of Ada applications software utilizing methods and procedures specifically tailored for this project.

The goal of this modernization effort (both Joint and Army) is to reduce the life cycle costs through reliable, portable, and reuseable code produced under strict configuration control utilizing the DOD-mandated Ada programming language [DOD87a, DOD84b]. AWIS expects nearly two million lines of code to be produced incrementally with phased releases to the multiple world-wide sites. Because there is insufficient direct experience in the management of such a program utilizing Ada under the guidelines of DOD STD-2167 [DOD 85], this contract has been on the leading edge of both Ada technology and management of that technology.

From a Program Manager's view point, there are four areas of importance required for forecasting, planning, and controlling a project to achieve the desired end product. These areas are resources, schedule, quality, and risk. Resources are those replenishable ingredients of people, equipment, facilities, and support (e.g., communications, supplies, etc.). Schedules deal with the one non-replenishable resource - time. Quality consists of an evaluation of how well the product conforms to the desired specifications. Risk is the degree of expected success in managing the first two factors in order to achieve the third. The common denominator across all factors is cost.

AWIS is in the process of developing methodology, collecting data, and performing analyses to develop Metrics to support the management and control of the above variables. Results to date are preliminary but the approach appears to have merit. They are presented to encourage discussion and invite the interchange of concepts and findings in this early but critical aspect of Ada development.

BACKGROUND

The World Wide Military Command and Control System (WWMCCS) serves the National Command Authority (NCA) and key commanders across a broad spectrum of planning and operational activities from day-to-day through crisis operations to conventional and nuclear war. The computers, software, and associated telecommunications form the backbone of the current WWMCCS ADP system. Its use is expanding both in breadth and depth of operational applications.

Modernization of the WWMCCS program officially began in November 1981 with the establishment of the WIS Joint Program Manager (JPM). The JPM is the focal point for coordination and control of all existing WWMCCS ADP requirements and upgrades. The overall goal of the WWMCCS Information System (WIS) program is to build an affordable system that makes maximum use of commerical off-the-shelf (COTS) hardware and software to provide responsive support to JCS validated operational requirements. The new system will allow for the future integration of state-of-the-art improvements, as well as provide system enhancements in survivability, DOD standard protocols, modularity, flexibility, sustainability, etc.

Army WIS (AWIS) is a part of, and an extension to, the larger joint modernization effort called WWMCCS Information System (WIS). As the Army portion of WIS, AWIS includes those service and command capabilities in support of the Joint and Army missions not provided by WIS. The Army maintains primary responsibility for implementing these supporting applications. AWIS will provide information to WIS and additional information handling capabilities at the WWMCCS sites managed by the Army. Therefore, AWIS, the Army subset of WIS, provides joint operational planning and execution capabilities to meet the command supporting requirements at the eight sites for which the Army is responsible.

The AWIS Software Development (ASD) effort will provide the applications software for Army and command subsystems that will support WIS. The applications software is being designed and implemented in Ada. The AWIS applications software will provide for command-supporting wartime and transition functions and will be resident on joint hardware at the appropriate locations.

The contract, being performed by TRW, Inc., is divided into two phases. Phase I is structured to provide a foundation for the full scale development of Ada software to replace the more than eight million lines of COBOL and FORTRAN software presently being operated at the Army supported WWMCCS sites. Phase I will acquire an Ada Programming Support Environment; develop plans, procedures, and a methodology for Ada development; establish training courses specific to AWIS; and build a pilot subsystem to ensure that the methodology, plans and procedures are correct. Phase II will use the approved methodology and procedures to implement the full-scale development of the estimated two million lines of Ada code. Phase I is projected to be completed in February 1988, with Phase II immediately following.

FOUNDATION

Metrics are the quantification of performance projected for or resulting from the management of resources, schedule, quality, and risk. Metrics provide both a prospective and retrospective measure of accomplishment. They are vital components of management on both the part of the Government and the contractor. Retrospective data provides a baseline for the next program phase. The more important use of metrics to a program manager is prospectively. It is in this mode of usage that forecasting, planning, and controlling can be exercised.

The challenge facing a program manager with Ada as the software development language is the sparseness of retrospective data. In recognition of this deficiency in the Ada environment, the Army included metrics development as one of the foundation tasks for Phase I of AWIS. Specific emphasis is given to developing an Ada Cost/Schedule Estimating Model [DEL88]. Initially it was envisioned that metrics would only be gathered in Phase I. However, it has been realized that additional data points are needed.

Key to improving software cost estimation capabilities is to collect data, data, data. This is the reason for extending the metrics task into Phase II. The Government has realized that the Phase I metrics tasks only provides a single data point. This point is only one of many that makes up a cost estimating curve. The exact shape of that curve can not be determined by only one point. More data is needed to determine whether the curve is a straight line or some other shape.

Prospective usage of metrics is most effective where the environment and methodology can be controlled and the cycle of activities repeats itself a sufficient number of times so that the metrics can be refined and applied. The longer the period of performance and the higher the number of cycle repetitions, the greater the benefits to be derived. AWIS Phase II has these characteristics.

Why bother with metrics at all? The following quote from Lord Kelvin sums it up.. "When you can measure what your are speaking about, you know something about it. When you are unable to use a quantitative description, then your knowledge is meager and unsatisfactory."

The fundamental goal of metrics is to gather data from an historical perspective, analyze the data, and formulate a methodology of predicting future performance of similar developments. In this project, the Government desires to prospectively determine the incremental costs of the full-scale Phase II effort. The Government needs to more accurately forecast the cost and schedule of this development project. Adequate funds must be programmed, and more importantly, defended to assure development completion. Once funds are budgeted, programmed and provided to the project office, then the amount of work to be tasked to the contractor can be properly scoped using the metric data. Schedules must be known to assure ease of transition at each of the Army supported WWMCCS sites. Many sites require long-term planning to assure proper integration of implementation scheduling.

Metrics data can also be used by the Army (Government) to evaluate how much IV&V effort is required for a particular development. Analysis of the data can help direct the proper application of the IV&V effort available. Metrics can then be used to determine when the IV&V effort has achieved "acceptable" results.

From the TRW (Contractor) point of view, metrics provide a baseline against which resource, requirements, schedules, and risks can be identified, corrective actions evaluated, performance measured, and success achieved. Quality may be evaluated prospectively. The availability of good metrics will permit TRW to respond quicker and more accurately to the development scheduled required by the Army. The resultant improvement in planning will yield better resource utilization and, therefore, lower costs.

Although there may be differences between the Government and the contractor in where sub-optimization can or should be achieved, the end goal of success is common and predominant. Effective utilization of metrics requires both the Government and the contractor to believe the viability of the metrics and be willing to use them as a significant basis for forecasting, scheduling, planning, and control. Metrics can help alleviate the fears and fantasy associated with Ada [BAM87].

Metrics are of little value without definition and control of the context in which they are developed. In recognition of this, the Army required several early products in Phase I to establish the proper foundation for Phase II. TRW also, recognized this and invested during Phase I in the creation of a processing and development environment sufficiently robust to support Phase II.

In the area of control, the objective is to establish a feed-forward mechanism rather than a feed-back mechanism of control. This provides the Program Manager with an ability to "steer" the project to the desired goal, rather than recovering after the fact.

Among the first products to be produced were a description of the design methodology and a Software Standards and Procedures Manual. Another early product was a study that led to the selection of an APSE, including the initial tool set. A risk management program, configuration control system, quality evaluation program, and metrics development program were all started very early in Phase I.

With all the components in place, a preselected subset of a Functional Description was implemented to verify the foundation and provide for appropriate refinements where the processes, procedures, and controls were found lacking. Strict configuration management control was exercised at all times.

The design, development, test, and delivery of Ada software may span a period from several months to several years depending on the size and complexity of the application. Metrics which are not available until after program completion are of little benefit. AWIS has chosen the seven software development phases of DOD STD-2167 to provide the first level of decomposition of activities and performance. The intent is that the Metrics at the end of the

Requirements Analysis, Preliminary Design, Detailed Design, Code and Unit Test, CSC Integration and Test, CSCI Integration and Test, and DT&E will provide the ability to evaluate performance on the phase being completed and adjust, as necessary, the expectation for subsequent phases.

When the retrospective data is meager, there is initially a need to "over-collect" data and perform analysis to discover the highly leveraged factors that should constitute the final Metrics set. As analysis determines low correlation of the factors to performance measurement and/or the ability to impact performance using those factors, they are discarded. Where two or more factors are interdependent and sensitive to the same management actions, they are either combined or the most appropriate factor is selected for continued data collection. AWIS is in the initial stages of this determination.

There is a need to have a high degree of automation in the Metrics program to assure timeliness of results and to minimize the cost of the Metrics activities on life cycle costs. AWIS has used available tools where ever possible to support the metrics activities. Since insufficient data points are available to determine the final metrics set, only a small number of specialized tools are currently being applied to AWIS.

Preliminary methodologies and results from this foundation phase are presented in two areas: Productivity and Quality. Continued refinements to the methodologies and procedures are obviously required and are on-going.

PRODUCTIVITY

It was decided early-on that the lack of global retrospective data for Ada developed under DOD STD-2167 guidelines required AWIS to collect and analyze detailed data in a variety of areas. To facilitate this data collection and analysis, a generic model was developed and is presented in Figure 1. The project was segmented into the seven software development phases described in DOD STD-2167. Using this generic model, a mini-model is developed for each of the phases. Each mini-model has its own set of algorithms and is constrained so that the output of one mini-model provides the input to the next mini-model.

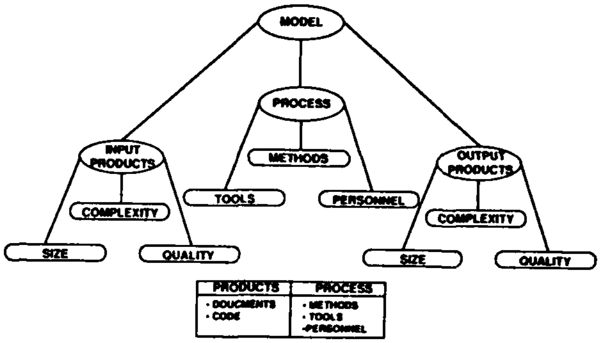


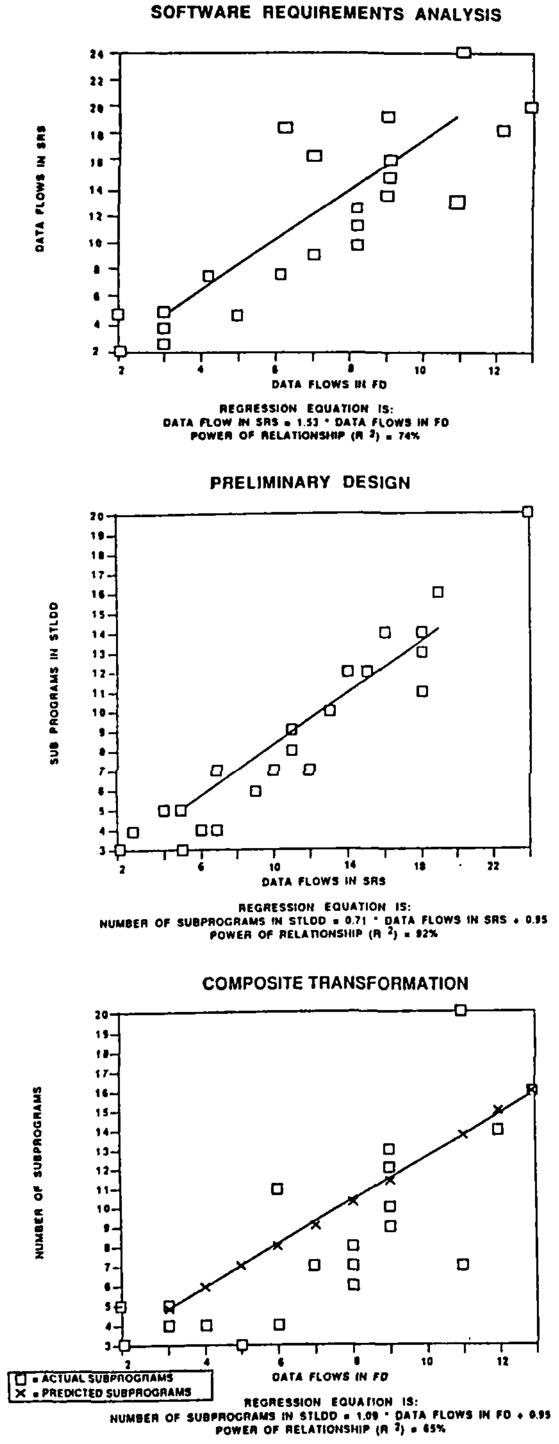
FIGURE 1 - METRICS MODEL

The Metrics model permits data to be collected at each of the development phases and still be aggregated over any contiguous number of phases. Thus, the resulting metrics at the end of any phase provide for retrospective analysis, adjustment, and a prospective look at succeeding phases of the total software development cycle.

In the first phase of software development, the input products are the System Specifications and the Functional Description (FD). The output of the first phase is the Software Requirements Specification. The Software Requirements Specification is also the input to the second phase which is Preliminary Design. The process flow continues in this manner through the total software development cycle.

The Functional Description provided to TRW by the Army describes the functionality of the system through a series of ADP events. The data flows in the FD associated with an event are translated into data flows for the SRS. The data flows lead to sub-programs as described in the Software Top Level Design Document at preliminary design. Figure 2 shows the correlation between these parameters on the selected subset of events implemented in Phase 1.

One of the goals of AWIS is to provide an Ada Cost/Scheduling Model. For most models that exist today, there is a requirement to estimate the lines of code or the number of units to be developed. This is one of the more imprecise requirements in the process. There has been evidence for some time [BEL 76] that over the life cycle of a software product, the number of "modules" has an impact on effort expended to maintain and enhance the product. Other product parameters may also have a bearing on life cycle costs.



SOFTWARE REQUIREMENTS ANALYSIS

REGRESSION EQUATION IS:
DATA FLOW IN SRS = 1.53 * DATA FLOWS IN FD
POWER OF RELATIONSHIP (R^2) = 74%

PRELIMINARY DESIGN

REGRESSION EQUATION IS:
NUMBER OF SUBPROGRAMS IN STLDD = 0.71 * DATA FLOWS IN SRS + 0.95
POWER OF RELATIONSHIP (R^2) = 92%

COMPOSITE TRANSFORMATION

REGRESSION EQUATION IS:
NUMBER OF SUBPROGRAMS IN STLDD = 1.09 * DATA FLOWS IN FD + 0.95
POWER OF RELATIONSHIP (R^2) = 65%

FIGURE 2 - METRICS CORRELATION

AWIS has the fundamental tool set necessary to support the collection on analysis of productivity metrics. This tool set will be augmented as the most meaningful metrics are determined and as automation is determined to be cost effective.

The Complexity Measures Tool (CMT), by EVB Software Engineering produces a count of the lines of code (Blanks, Comments, Executable Lines, Data Declarations, and Total Physical Lines). Additionally, it computes the McCabe cyclomatic complexity metric and the fifteen Halstead Software Science Metrics.

A tool has been developed by AWIS to tally numbers of Ada type declarations, Ada subprogram declarations, and "WITH" clauses. The counts from this tool also supports the determination of Ada design complexity indicators.

The accounting utility of the development processor is used to collect data on AWIS CPU usage, connect time, storage, page faults, and I/O. These data support both the analysis of development resources consumed and the determination of sizing and timing data.

AWIS uses Change and Configuration Control by SOFTOOL Corporation in support of basic configuration management.

Lotus 1-2-3 on the IBM PC supports analysis and reporting. STATPAK (PC) provides regression analysis.

AWIS collects and controls costs and schedules using Cost//Schedule Control System Criteria delineated in DODI 7000.2. The Work Breakdown Structure provides the cost accumulation and scheduling framework and supports the calculation of productivity measures. The WBS is consistent with the selection of the DOD STD-2167 software development phases for productivity calculations. Modifications and extensions of the WBS have been required at lower levels for more detailed cost collection.

The establishment of schedule networks and critical path analysis is further supported by the use of ARTEMIS.

No code is generated during the requirements phase. However, because of the use of ADL for design, lines of code are generated during preliminary design. AWIS is developing a profile of productivity on a weekly basis. An example is shown in Figure 3 for the period from the start of development to CDR. Attention is called to the decrease occurring at 226 Days After Contract. This decrease resulted from a simplification implemented during design which permitted the removal of code. The reduction in productivity 268 DAC resulted from no code being produced during CDR. Similar anomalies have provided insight into the impact of various technical and management decisions. The full impact of such visibility is as yet unknown, but value has already been realized.

TIME	DAC	Blank Lines	Structured Comments	Executable Lines Of Code	Data Declarations	Physical Lines	SLOC	Man-mths To Date* (Software & Data Base Related Activities)	Cumulative Productivitye (SLOC/Man-Months)
CONTRACT AWARD	0	0	0	0	0	0	0	0	0
SSR	106	0	0	0	0	0	0	26.44	0
PDR	194	3052	10387	455	5298	21742	5753	45.01	128
	219	4650	14154	776	6016	27505	6792	54.43	125
	226	5022	16123	813	5572	30542	6385	56.64	113
	233	4575	14959	740	4981	27953	5721	59.40	96
	240	4941	14748	676	5496	28098	6172	62.43	99
	247	5339	16582	955	5838	31452	6793	64.92	105
	254	6572	32491	1431	7945	52403	9376	65.90	142
CDR	261	6855	36670	1518	8216	57373	9734	67.11	145
	268	6855	36670	1518	8216	57353	9734	69.95	139
	275	6961	35586	1816	8059	56952	9875	72.11	136

*Man-Hours include time spent on non-deliverable software tools developed

FIGURE 3 - PRODUCTIVITY

Productivity increases to a peak at the completion of coding then decreases through unit test and subsequent activities. When sufficient data points are available, the profiles will be used to evaluate current progress against estimates for developing end-item software. Significant variation in profiles and levels of productivity will be used as triggers for futher management evaluation.

QUALITY

Quality evaluation is a requirement of the AWIS program. The areas of concern are shown in Figure 4. Again, it was decided that data should be collected against the software development phases as described in DOD-STD-2167. It was determined that the Automated Measurement System (AMS) [RAD85] was the best tool available for use as the fundamental tool for this analysis. AMS instruments the RADC Software Techology for Adaptable, Reliable Systems (STARS) framework for the 13 quality factors shown in Figure 4. AMS is used by the quality evaluation team to assess the software quality at the end of each software development phase.

Design - How valid is the design?
- Correctness
- Maintainability
- Verifiability

Performance - How well does it function?
- Efficiency
- Integrity
- Reliability
- Survivability
- Usability

Adaptation - How adaptable is it?
- Expandability
- Flexibility
- Interoperability
- Portability
- Reusability

FIGURE 4 - QUALITY CONCERNS

The methodology employed appears to have merit in "scoring" each of the "ilities". It does not yield quantified results on such parameters as reliability, survivability, efficiency, etc. Significant tailoring to the question set has been required for an Ada/Object Oriented Design environment. Problems have been encountered with some of the algorithms. Also, as a new tool and methodology, some problems have occurred with TRW's implementation.

During the first phases, the quality evaluation scoring lacked sufficient discrimination in some of the metrics, e.g., scores of 1.0. The question sets will be reviewed for the introduction of more sensitivity where possible.

Figure 5 shows the population from which quality evaluated metrics were developed during the Detailed Design Phase.

	Number		Metrics
Structure	Total	Distinct	Computed
CSCI	1	1	57
TLCSC	5	4	10
LLCSC	32	21	10
UNITS	216	22	118

FIGURE 5 - CDR METRICS POPULATION

Analysis of the metrics yielded the scoring shown in Figure 6. Several anomalies were evident. As an example, the reliability score was a surprise. Subsequent analysis identified several problems with the tool and/or its application. Several questions assumed strict design rather than Object Oriented Design. Several questions relative to Units were misapplied at a higher level. Several formulas were found to be faulty.

IMPORTANCE	FACTORS	DETAILED DESIGN SCORE
HIGHEST	CORRECTNESS	0.92
	VERIFIABILITY	0.72
	RELIABILITY	0.64
	USEABILITY	0.99
	REUSEABILITY	0.72
	PORTABILITY	0.89
	MAINTAINABILITY	0.79
	EFFICIENCY	0.81
	INTEGRITY	N/A
	EXPANDABILITY	0.61
	FLEXIBILITY	0.65
	INTEROPERABILITY	0.82
LOWEST	SURVIVABILITY	0.87

FIGURE 6 - CDR QUALITY FACTOR CALCULATION

The question set was altered and the application adjusted, and the formulas corrected. Re-evaluation yielded the results shown in Figure 7. Work continues to evaluate the other factors.

ACCURACY (AC)

= N/A THIS PHASE

ANOMOLY (AM)

= AVE (ERROR TOLERANCE/CONTROL = 1.0
HANDLING IMPROPER INPUT DATA = 0.94
HANDLING COMPUTATIONAL FAILURES = 0.97)
= 0.97

SIMPLICITY (SI)

= AVE (DESIGN STRUCTURE = 0.80
STRUCTURED LANGUAGE = 1.00
CODING SIMPLICITY = 0.95
SPECIFICITY = 0.97)

RELIABILITY (RE)

= AVE (AC, AM, SI) = 0.95

FIGURE 7 - REVISED CDR RELIABILITY FACTOR

Conclusions at this stage are that the concepts are good. Considerable work is required on both the questions and the formulas. The results should be interpreted as grades only. AMS provides more available metrics than are required. Analysis for eliminating some metrics and reduction of the question set for others is underway.

This tool is less mature than was expected, but the methodology appears to offer long term benefits. AMS should become more useful as its usage by other projects increases. As a scoring system, it still lacks a solid baseline against which to interpret the "goodness" or "acceptability" of results in the achievement of project requirements. It does not replace the need for the more conventional quality assurance metrics but does appear to augment the prospective value of indicators. Actual values for the factors still need to be derived during testing and usage of the product.

CONCLUSION

The management of "first" projects represents some unique and many distinct challenges. The AWIS software development project is facing those unique challenges head on. In the area of metrics, a program is well in place but has only scratched the surface in the resolution of this elusive problem. Advances have been made and set backs have been encountered. The net thus far has been positive. Work continues because the cost benefit potentials are high.

With all the attention being given the subject, the availability of global Ada metrics should grow rapidly. In two or three years, a project such as AWIS will no longer be categorized as a "firsts" project.

ACKNOWLEDGMENT

The authors wish to acknowledge the work done by Dr. J. J. Logan and A. Kulkarni, of TRW, in establishing and evaluating the productivity metrics. D. DeHaven, of TRW, and D. Norris, of Harris Corporation, were major contributors to the adaptation, implementation, and analysis associated with AMS.

BIBLIOGRAPHY

[BAM 87] J. Bamberger, What Every Good Manager Should Know About Ada, Conference Proceedings: National Aeronautics and Electroncis Conference (NAECON '87), Dayton, Ohio, May 18-27, 1987.

[BEL 76] L. A. Belady & M. M. Lehman, A Model of Large Program Development, IBM Systems Journal, 1976.

[DEL 88] LTC R. P. Delaney, U. S. Army, L. F. Summerill, Management Perspective of a Front Running Ada Project, Conference Proceedings: U. S. Army Information Systems Engineering Command Technology Strategies '88. (TS'88) Conference, Alexandria, Virginia, February 9-12, 1988.

[DOD85] Department of Defense. Defense System Software Development: DOD-STD-2167. United States Government, 4 June 1985.

[DOD87a] Department of Defense. Computer Programming Language Policy. DODD 3405.1. Department of Defense. March 30, 1987.

[DOD87b] Department of Defense. Use of Ada in Weapon Systems. DODD 3405.2. Department of Defense. April 2, 1987.

[RAD85] RADC-TR-85-37, Volume III, Final Technical Report, February 1985. 'Specification of Software Quality Attributes, Software Quality Evaluation Guidebok', Rome Air Development Center AF Systems CMD, Griffiss AFB, New York, 13441-5700.

Experience Using an Automated Metrics Framework in the Review of Ada Source for WIS

J. D. Anderson and J. A. Perkins

Dynamics Research Corporation
60 Frontage Road, Andover, MA 01810

ABSTRACT

Analysis of the WIS Ada source code involved applying an automated, hierarchical, Ada-specific software metrics framework to approximately 200,000 lines of Air Force-supplied Ada source. The purpose of the analysis was to aid the Air Force in identification of the characteristics of the code that detract unnecessarily from reliability, maintainability, and portability. The software was analyzed during the initial phase of code development to insure that sufficient time would be allotted for the elimination of undesired characteristics.

DRC's Ada metrics framework measures three software factors, six software criteria, and 150 software metric elements, where each metric element relates a software quality principle to the use of specific features of the Ada language.

The analysis of the Air Force-supplied Ada source involved: 1) automated calculation of metric scores for the supplied source, 2) human analysis of the metric scores to determine those characteristics that augment or attenuate quality and to formulate recommendations on how to enhance quality, 3) modification of two modules of the supplied source to illustrate the impact of our recommendations, and 4) reporting of the findings to the Air Force.

KEYWORDS

software metrics, software quality, Ada, software principles, software tools

ACKNOWLEDGMENT

The analysis discussed in this paper was funded under contract number F19628-86-D-0006/0012 by the Electronics Systems Division of the Air Force Systems Command.

Ada is a registered trademark of the U.S. Government(AJPO).
ADAMAT is a trademark of Dynamics Research Corporation.

1. INTRODUCTION

Dynamics Research Corporation (DRC) has developed an Ada Measurement and Analysis Tool, ADAMAT, designed to use software metrics analysis to improve the quality of Ada software. The metrics in the framework used by ADAMAT are automated, hierarchical, and Ada-specific. Each low-level metric in the framework is: 1) based on an underlying software quality principle, 2) defined in terms of specific features of the Ada language, and 3) documented to indicate both the rationale for the metric and the method for improvement of software when the quality problem related to the metric is detected [Keller 85, Perkins 85].

In previous studies, involving analysis of Ada code developed at DRC [Perkins 86] and Ada code supplied by the Naval Underwater System Center [Perkins 87], we illustrated the usefulness of automated Ada-specific metrics for the detection of quality problems; the identification of specific Ada features, indicating where training of Ada software personnel is required; and the improvement of quality of Ada software. In this paper, describing DRC's metric-based analysis of Ada source performed for Air Force Electronic Systems Division (ESD), we discuss the effectiveness of metrics as an aid to reviewing the quality of large Ada code segments.

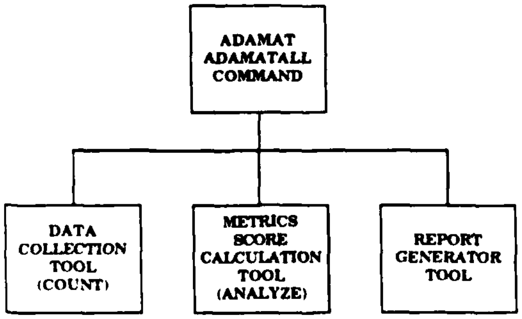


FIGURE 1 ADAMAT COMPONENT DIAGRAM

Our work for ESD involved collecting metric scores for approximately 200,000 text lines of Air Force-supplied Ada source, analyzing these metric scores to determine those characteristics of the source that augment or attenuate quality, and reporting our findings and recommendations to the Air Force. This work was performed during the early stages of code development to provide ample time for the Air Force and the software contractor to review our report and react to the recommendations. This helped to insure that those characteristics of the source considered by the Air Force as unnecessarily detracting from reliability, maintainability, and portability could be eliminated from the software by the software contractor prior to the start of the testing phase. Our analysis was limited to a fixed time period in the life cycle of this source, with no regard to the previous history of the source.

For this study, the meaning of the term "quality" is limited to three software factors: reliability, maintainability, and portability. These factors are measured with respect to the following six criteria:

- o Anomaly Management,
- o Independence,
- o Modularity,
- o Self Descriptiveness,
- o Simplicity, and
- o System Clarity,

where each of these criteria is further defined in terms of low-level metric elements.
The investigation involved the following steps:

- o Automated data collection (Section 2.1) and metric score calculation (Section 2.2) for each of the files constituting the Ada source, and the reporting of metric scores (Section 2.3) in three views: for each of the 335 files, for the 75 distinct higher-level groupings of those files, and for the composition of all the files,

- o Recommendations on how to improve the quality of the Ada source with respect to each of the six software criteria (Section 3.1),

- o Analysis to determine those characteristics of the Ada source that enhance these criteria and those that raise quality concerns (Section 3.2), and

- o Modification of two selected modules to illustrate the impact of incorporating the complete set of recommendations with respect to reliability, maintainability, and portability. (Section 3.3).

2. AUTOMATED COLLECTION AND CALCULATION

Our metric analysis started with receipt from the Air Force of a tape containing 335 separate files of compilable Ada source. Our initial objective was to calculate and report metric scores for each of these 335 files, for the 75 distinct groupings of these files, and for the Ada source as a whole (as a "336th file). The production of the desired 411 metrics reports required three steps: 1) collecting data items for each file, 2) calculating metrics scores for each file and each group of files, and 3) producing the hard-copy reports for each resulting set of metric scores (Figure 2). Each of these steps was automated using the three principal components of ADAMAT - COUNT, ANALYZE, and REPORT (Figure 1).

```
GOOD     TOTAL    METRIC NAME
*        *        anomaly management
*        *          user_exceptions_raised
13278    14947      user_types
*        *          applicative_declarations
*        *          constrained_variant_records
*        *          loop_normal
366      470        constrained_subtype
*        *          default_initialization
*        *        independence
*        *          no-implementation_defined_attributes
*        *          no_pragma_interface
*        *          no-implementation_dependent_pragms
572      594        no_component_clause_for_record_types
*        *          no_length_clause_for_storage_size
*        *          no_length_clause_for_size
2        10         numeric_type_declarations
*        *          numeric_constant_declarations
*        *          no_pragma_pack
*        *          macharithindependence
*        *          no_min_int
*        *          no_max_int
*        *        modularity
*        *          no_multiple_type_declaration
*        *          block_declarations
*        *          limited_size_profile
0        391        no_variable_declarations_in_specifications
*        *          user_defined_operations
95       992        private_types
*        *        self_descriptiveness
27068    28153      no_predefined_words
*        *          number_of_commented_declarations
*        *          number_of_commented_statements
1463     2559       number_of_commented_bodies
*        *          number_of_commented_specifications
*        *        simplicity
3707     13347      declarations_contain_literals
*        *          array_range_explicit
*        *          subtype_declarations_explicit
175      220        array_type_declarations_explicit
*        *        system_clarity
*        *          qualified_aggregate
*        *          named_aggregate
*        *          named_exits
*        *          module_end_with_name
34       262        named_blocks
*        *          named_loops
*        *          single_object_declaration_lists
*        *          limited_private_access_types
3055     3120       no_default_mode_parameters
*        *          for_loops_with_type
*        *          no_while_loops
*        *          expressions_parenthesized
*        *          non_negated_boolean_expressions
```

FIGURE 2 ABBREVIATED METRICS REPORT FOR COMPOSITION OF ALL FILES

2.1 COLLECTION OF DATA ITEMS

The first step in the automated portion of the metrics analysis is collection of data item counts. "Data item" is the classification given to the lowest elements in DRC's metrics hierarchy (Figure 3). COUNT parses a file of compilable Ada software to produce a set of data item counts for that code. Each data item count is an integer value indicating the number of occurrences in the source of a specific feature of the Ada language. The data items are actually collected by module for each file; as a result, later recommendations may focus on the specific modules of the software. "Module" is used here to mean the specification or body of a generic or non-generic function, procedure, task, or package.

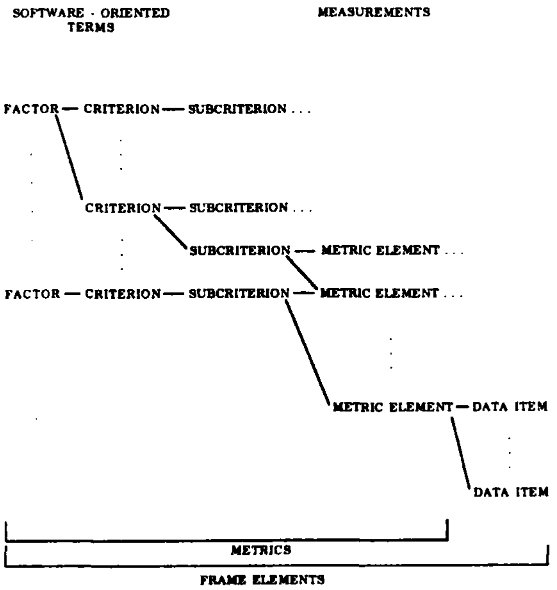

SOFTWARE - ORIENTED TERMS
MEASUREMENTS
FACTOR — CRITERION — SUBCRITERION . . .
CRITERION — SUBCRITERION . . .
SUBCRITERION — METRIC ELEMENT . . .
FACTOR — CRITERION — SUBCRITERION — METRIC ELEMENT . . .
METRIC ELEMENT — DATA ITEM
DATA ITEM
METRICS
FRAME ELEMENTS

FIGURE 3 HIERARCHICAL STRUCTURE OF THE METRICS FRAMEWORK SUPPORTS MEASUREMENT OF SOFTWARE QUALITY

Producing the 335 sets of data item counts required for this work necessitated 335 runs of COUNT; each of the 335 files was processed individually (Figure 4).

INPUT	COMPONENT	OUTPUT
file 1	count	d1, set of data item counts for file 1
file 2	count	d2, set of data item counts for file 2
. . .	. . .	. . .
file 335	count	d335, set of data item counts for file 335

FIGURE 4 COLLECTION OF DATA ITEMS

2.2 CALCULATION OF METRIC SCORES

The second stage in the automated portion of metric analysis is calculation of metrics scores. "Metric" is the classification given to all elements in the metrics hierarchy that occur at any level above data items. Metrics are subdivided into factors, criteria, subcriteria, and metric-elements (See Figure 3). ANALYZE calculates a single set of metric scores from one or more sets of data item counts, where each metric score is an ordered pair (good, total) of integer values indicating the number of "proper" occurrences and the number of "total" occurrences of specific features of the Ada language present in the source. This ability to process multiple sets of data item counts allows the calculation of metric scores for a grouping of files without requiring the files to be composed at the source level.

Producing the 411 sets of metric scores required for this work necessitated 411 runs of ANALYZE. The data item counts for each of the 335 files, the composition of the data item counts for the files of each of the 75 distinct groupings, and the composition of data items of all 335 files were processed to produce the sets of metrics scores (Figure 5).

INPUT	COMPONENT	OUTPUT
d1	analyze	m1, metrics scores for file 1
d2	analyze	m2, metrics scores for file 2
. ..	. . .	. . .
d335	analyze	m335, metrics scores for file 335
d1 . . dj	analyze	n1, metrics scores for group 1
dj+1 . . dk	analyze	n2 metrics scores for group 2
. .	. . .	. . .
dx+1 . . d335	analyze	n75 metrics scores for group 75
d1 . . . d335	analyze	c, metrics scores for composition of all files

FIGURE 5 CALCUATION OF METRICS SCORES

2.3 PRODUCTION OF HARD-COPY REPORTS

The third stage in the automated portion of metric analysis is production of the hard-copy reports containing the metrics scores. REPORT produces a single formatted, hard-copy, metrics report from one or more sets of metric scores, and the corresponding data item counts. The ability to process multiple sets of metric scores allows the comparison of sets of metric scores in a single report. The form and content of the report may be controlled by the user/analyst.

We chose to report only those metric scores where the number of "proper" occurrences was less than the number of "total" occurrences, and not to report the individual data item counts. These restrictions allowed the metric-based human analysis that followed to focus quickly on "improper" occurrences, and guaranteed that the size of any report would be independent of the number of modules in a given file (recall that although data items are collected on a module basis, the metric scores are calculated on a file basis). The information lost by omitting the data item counts from the report is relatively small since numbers of proper and total occurrences for a metric-element are made up of the counts of the associated data items. 411 runs of REPORT produced the 411 metric reports required by this work, in other words, one run for each set of scores produced by ANALYZE (Figure 6).

INPUT	COMPONENT	OUTPUT
m1	report	metrics report for file 1
m2	report	metrics report for file 2
. .	. . .	. . .
m335	report	metrics report for file 335
n1	report	metrics report for group 1
	. . .	. . .
n75	report	metrics report for group 75
c	report	metrics report for composition of all files

FIGURE 6 PRODUCTION OF HARDCOPY REPORTS

To produce comparison reports, for example the metric scores for each grouping compared to the scores of the individual files in that grouping, would have required an additional 76 runs of REPORT; one run for each grouping (Figure 7).

INPUT	COMPONENT	OUTPUT
d1, . . dj, n1	compare	metrics report for comparison of group 1 to file 1 . . file j
. .	. . .	. . .
dn+1 . . d335, n75	compare	metrics report for comparison of group 75 to file n+1 . . file 335
c, n1 . . n75	compare	metrics report for the comparison of group 1 . . group 75 to the Ada source as a whole

FIGURE 7 PRODUCTION OF COMPARISON REPORTS

3. METRIC-BASED HUMAN ANALYSIS

This section discusses how human analysis of the generated metric scores provided a means of outlining metric-by-metric potential non-adherence to accepted software quality principles and of identifying overall characteristics of the code which augment or attenuate quality.

3.1 METRIC-BY-METRIC ANALYSIS

The metric scores generated for the composition of an entire set of supplied files (ALL report) was reviewed, metric-element by metric-element. Since each metric-element of our framework is associated with a software quality principle, the metric scores provided a means of measuring the extent to which the code adheres to these principles.

Of the 153 metric-elements examined, 45 metric scores indicated a level of potential non-adherence sufficient to warrant further analysis. Of these, 12 metric scores, two for each of the six criteria, have been chosen for discussion in this paper. Each discussion contains a definition of the metric, the rationale for the metric, and the actual score the supplied code achieved.

The metric reports for each of the individual files allowed us to quickly locate code containing actual examples of non-adherence. The analysis of these code segments involved trying to determine the reason for non-adherence, the negative effects of non-adherence if any, and making sample modifications to the code to see the actual effects of obtaining adherence to the criteria. Each of the chosen metric elements are discussed below according to the criterion with which they are associated.

For anomaly management,

o user_types: the proportion of type or subtype references to user-defined types rather than pre-defined types.

Variables, constants, and parameters were declared in terms of system-defined types. Declaring objects in terms of system defined types is not recommended because the effectiveness of strong type checking and range checking features of Ada is reduced. The score for this metric over all files was (13278, 14947) or 89%.

- constrained_subtypes: the proportion of subtype declarations containing a constraint. Subtypes were declared without constraints. In most instances, declaring a subtype without a constraint provides no advantages beyond those already provided by the base type, in terms of strong type checking or range checking. This again reduces the effectiveness of the range checking features of Ada.

  As a related consequence, it should be noted that frequent use of such subtyping without range may indicate that a parent type is being used to group conceptually different but structurally similar objects, resulting in loss of advantages which could be gained from Ada's type checking. The score for this metric was (366, 470) or 78%.

For Independence,

- numeric_type_declarations: the proportion of numeric type declarations which are declared without using an associated explicit type.

  Most of the numeric type declarations reference system-defined types. References to the system defined types force the compiler to use system-specific representation which may differ from machine to machine. The score for this metric was (2,10) or 20%. The occurrence of only 10 numeric type declarations in a program of this size is an indication that the numeric types supplied by package STANDARD are being overused. In fact, most of the references to system types recorded by the user-types metric are because of the reliance on type INTEGER.

- no_component_clause_for_record_types: the proportion of record type declarations that do not contain a component clause.

  Component clauses create dependency on the word-size of the target machine. In those cases where component clauses are justified, comments should be added to indicate the precise reason for their use. The score for this metric was (572, 594) or 96%.

For Modularity,

- no_multiple_type_decls_in_package_spec: the proportion of package specifications containing less than two type declarations.

  The presence of multiple types in a package may indicate that conceptually different objects are being defined in a single package. The need to reference any of these objects results in the ability to access all of these objects even when such access is not desired. The score for this metric was (93,153) or 61%.

- private_types: the proportion of types declared in the non-private part of the package specifications that are declared as private or limited private.

  Non-private composite types in package specifications usually indicate a lack of information hiding. Every user of the package is dependent upon the underlying data structure used to represent objects. Consequently, any changes made in the data structure will require changes by the users. Score for this metric was (95, 992) or 10%.

For Self_Descriptiveness,

- no_predefined_words: the proportion of names for packages, subprograms, types, subtypes, blocks, loops, constants, variables, numbers, parameters, exceptions, enumeration literals, loop parameters, entries, and components that are not predefined names.

  The reuse of a system-supplied name is not recommended because the reader of the code may be confused as to whether the name refers to a user-defined or system-defined object. Score for this metr'~ was (27068, 28153) or 96%.

- number_of_commented_bodies: the proportion of package bodies, task bodies, subprogram bodies, subunits, and body stubs that are commented.

  Lack of commenting of bodies increases the difficulty of understanding the functionality of the software. Score for this metric was (1463, 2559) or 57%.

For Simplicity,

- o declarations_contain_literals: the proportion of referenced numeric literals referenced in constant declarations and type declarations

  Using literals in the non-declarative portion of a program reduces clarity and increases the likelihood that a change to a literal will not lead to the appropriate modifications of other occurrences of that literal or that changes to identical literals may result in unintended changes to different objects. Score for this metric was (3707, 13347) or 28%.

- o explicit_array_types: the proportion of array type declarations which are non-anonymous.

  An anonymous array type declaration is of the form: "<identifier_list> : array <index_definition> of <subtype_indication> ..". Declaration of an implicit array type eliminates the ability to declare parameters, constants, and multiple variables of that type. Score for this metric was (175, 200) or 80%.

For System Clarity,

- o named_blocks: the proportion of blocks that are named.

  Lack of block identification results in structural, rather than a declared, association of the block BEGIN to block END. Score for this metric was (262,776) or 34%.

- o default_mode_parameters: the proportion of IN or default mode parameters to a procedure explicitly specified as IN mode.

  The lack of explicit specification of a parameter mode increases the likelihood that a parameter, intended to be OUT or IN OUT, is defaulted to IN mode. The score for this metric was (3055,3120) or 98%.

3.2 IDENTIFICATION OF CHARACTERISTICS

Our analysis of individual metric scores allowed the identification of overall characteristics of the code which augment or attenuate quality.

3.2.1 AUGMENTATION OF QUALITY

The following characteristics that augment quality were identified:

Our metrics for anomaly management showed that the software does not use the PRAGMA SUPPRESS, and does use the exception mechanism and type mechanisms of Ada.

The independence metrics indicated that the software does not use machine code statements, address clauses, alignment clauses,or floating point or fixed point types.

The scores for modularity showed that the library and package mechanisms are effectively used, and that subprograms are parameterized.

The self-descriptiveness metrics indicated that the identifiers have meaningful names, and that package specifications are well-documented.

The simplicity metrics showed that GOTO and ABORT statements are not being used, and the number of branches and level nesting within subprograms are not excessive.

The system clarity metrics indicated that the qualification and naming mechanisms of Ada are being used.

3.2.2 ATTENUATION OF QUALITY

The following characteristics that attenuate quality were identified:

The software is heavily dependent on the Ada system-defined integer types. In most cases, this dependency seems unjustified. Replacing the use of system-types by user-defined types is recommended.

The software is heavily dependent on implementation-defined pragmas and attributes, and on compiler and machine-dependent clauses and pragmas. In most instances, these dependencies seem justified and are well isolated. However, the occurrences of such features are well commented to indicate the precise reason for their use.

The software is heavily dependent on package specifications providing access to system-supplied operators. In many instances, composite types are not declared as PRIVATE. This extensive dependency of "WITHers" of packages on the underlying data structures seems unjustified. Composite types should be declared as PRIVATE.

The software is heavily dependent on a few packages that declare many types, In general, the declaration of so many types in a single package seems unjustified. These packages should be decomposed into smaller packages.

The software is heavily dependent on numerical literals in the executable part of the software. There seems to be no justification for this. Constants should be declared to represent these literals.

The software is heavily dependent on implicit types and subtypes. In general, there seems to be no justification for this situation. Explicit types and subtypes should be declared.

In most instances, the package specifications and bodies are well commented. However, there are many uncommented procedures, functions, and tasks. More commenting is recommended.

In general, the delineation of the structure of the supplied software is not difficult. However, more extensive use of qualification, naming of structures, and parenthesized expressions would be beneficial.

3.3 MODIFICATION OF SELECTED MODULES

Two modules of the supplied source were modified using the findings of the metric analysis. The modifications focused on the 45 metric scores warranting further analysis; however, non-adherence to principles associated with other metrics was addressed.

Most of the changes were straight-forward and required very little modification to the basic structure of the code. However, two of the most interesting changes did require structural modification.

The first instance of such change occurred in association with a loop, which contains references to 6 variables. One of these variables was accumulating the sum. One was used to hold the sign. The other four were assigned values that depended only on the current iteration of the loop. The original structure of the module required careful analysis to determine that 1) the value of the sign was evaluated during the first iteration of the loop, unchanged during successive iterations, and referenced during each iteration and 2) the current values of the other four variables were not in any way effected by their previous values. To enhance clarification, the code was modified by placing a block inside the loop and declaring these four non-recurrent variables there.

The other instance of such a change involved the functions inside a package body. As the suggested modifications were added to each of the functions within the package body, it became apparent that each of the functions was of the same general form, that of performing a search. If the search succeeded, an index indicating the position of the desired element was returned; otherwise an exception was being raised. The modification consisted of creating a generic search function with two generic parameters. The first parameter was the type of the index and the second was a function which evaluated the search condition based on the value of the index. All of the original functions were modified to define a function for the search condition, to instantiate the generic function, and to return the index value of this instantiated function.

4. REPORT DELIVERED TO AIR FORCE

The recommendations of the metric-based human analysis and and the hard-copy metric reports generated by ADAMAT were delivered to the Air Force, thereby allowing our findings to be integrated into their review process.

The Air Force also received a copy of the ADAMAT Reference Manual, our guide to the interpretation of the metric scores. Figures 8 and 9 show actual pages from this manual.

4.1 REPORT OF HUMAN ANALYSIS

The report of the metric-based human analysis of the Ada source contains three major segments. The first segment summarizes the characteristics of the overall code that augment or attenuate quality and the metric scores that support these findings. Figures 10, 11, and 2 show abbreviated versions of this summary.

The second segment contains an explanation of each of the metric scores that contributed to the summarized findings. The form of these explanations was similar to that of the reference manual; however, the examples were taken directly from the supplied software. Figures 12 and 13 show such explanations.

The third segment contains the modified versions of the two selected modules. The modifications recommended by the analysis are incorporated into these modules. These modifications were performed specifically to demonstrate the effect that our suggestions would have on the code; any actual modifications of these modules or any other modules of the supplied software are the responsibility of the software contractor.

NO_MULTIPLE_TYPE_DECLARATIONS

The proportion of package specifications containing no more than one type declaration.

In some cases, the presence of multiple type declarations in a package specification may indicate definition of multiple objects in a single package. The need to reference any of these objects will result in the ability to access all of the objects even when such access is not desired. In other cases, the presence of multiple types may indicate definition of a single object structurally too complex for representation by a single data type. The object represented may require operators not directly supported by any type and may not require all the operators supported by the types.

In some cases, multiple type declarations in a package specification should be replaced by multiple package specifications that each declare a single type. In other cases, one of the multiple type declarations should be defined to be private and others should be moved to the private part of the package specification.

For this metric, the following has a score of (0,1).

```
package stack_package is
   type stack_range_type is range 1 . . max_stack_size;
   type stack_array_type is array (stack_range_type) of
element_type;
   type stack_type is
      record
         stack_array_fld:   stack_array_type;
         stack_index_fld:   stack_range_type;
      end record;
```

The following has a metric score of (1, 0).

```
package stack_package is
   type stack_type is private;
   procedure pop_stack
      (stack:  in out stack_type;
      element:  out element_type);
   procedure push_stack
      (stack:  in out stack_type;
      element:  in element_type);
private
   type stack_range_type is range 1 . . max_stack_size;
   type stack_array_type is array (stack_range_type) of
element_type;
   type stack_type is
      record
         stack_array_fld:   stack_array_type;
         stack_index_fld:   stack_range_type;
      end record;
```

FIGURE 8 METRICS DEFINITION

NUMERIC_TYPE_DECLARATIONS

The proportion of numeric type declarations which are declared without using an associated explicit type:

The use of an explicit type forces the compiler to use a system dependent type. When no type is explicitly used a declared object is of a universal type. This improves the portability and generality of the program concerning a declared object.

Do not use explicit type when declaring a numeric type.

For this metric, the following has a score of (0,1).

```
type my_integer is new integer;
```

The following has a metric score of (1, 0).

```
type my_integer is range - 1E6 . . 1E6;
```

FIGURE 9 METRICS DEFINITION

CHARACTERISTICS THAT ENHANCE QUALITY

The software does not suppress the error-detection mechanism of Ada by using PRAGMA SUPPRESS.

The software does use the EXCEPTION mechanism of Ada. Exceptions are being declared, raised, and handled.

The software does not use machine code statements, address clauses, or alignment clauses.

The software does not use floating-point or fixed-point types.

The software does use the PACKAGE mechanism of Ada. Only types, constants, variables, procedures, functions, and tasks relating to the same high-level conceptual object are encapsulated within a single PACKAGE.

The software limits the use of the "WITH" and "USE" mechanism of Ada. Packages are "WITHed" only when needed.

The software effectively uses the "TYPE" mechanism of Ada. Enumeration, array, record, variant record, and access types are used when required.

Functions and subprograms in the software are parameterized.

The identifiers used in the software are given meaningful names.

The PACKAGE specifications in the software are well documented as to intent.

The software does not use GOTO or ABORT statements.

The number of branches and the level of nesting within modules of the software are not excessive.

The software does use the qualification and naming mechanisms of Ada. References to elements in "WITHed" PACKAGES are often qualified; references to components of aggregates are often named.

FIGURE 10 AUGMENTED QUALITY LIST

CHARACTERISTICS THAT RAISE QUALITY CONCERNS

The software is heavily dependent on the Ada system-defined integer types. In most cases, this dependency seems unjustified. Replacing system-defined types by user-defined types would increase the effectiveness of the strong type-checking capability of Ada, in the detection of unintended transfer of values between conceptually-different objects. User-defined types should be declared to replace these references to system-defined types.

The software is heavily dependent on implementation-defined pragmas and attributes, and on the compiler and machine-dependent clauses and pragmas. In most cases, these dependencies seem justified and are well isolated. However, for each occurrence of these features, a comment should be added justifying the precise reason for its use.

The software is heavily dependent on package specification providing access to system-supplies operators. In many instances, composite types are not declared as PRIVATE. This extensive dependency of "WITHers" of PACKAGEs on the underlying data structures seems unjustified. Composite types should be declared as PRIVATE.

The software is heavily dependent on a few packages that declare many types, thereby creating coupling between conceptually-different objects. In general, the declaration of so many types in a single package seems unjustified. These packages should be decomposed into smaller packages.

The software is heavily dependent on numerical literals in the executable part of the software. In most cases, this dependency is unjustified. Constants should be declared to represent these literals, so that multiple uses of the same literal for different purposes are not confusing.

The software is heavily dependent on implicit types and subtypes. In most cases, this dependency is unjustified. Explicit types and subtypes should be declared to replace the implicit types and subtypes, so that references to these types are possible.

In most instances, the package specifications and bodies are well commented. However, there are many uncommented procedures, functions, and tasks. Moreover, comments concerning the declaration of variables and constants or the reasons for transfer of control are often lacking.

In general, delineation of the software structure is not difficult; however, more extensive use of qualification, naming of structures, and parenthesized exoressions would be beneficial.

FIGURE 11 ATTENUATED QUALITY LIST

AVOID MULTIPLE TYPE DECLARATIONS IN PACKAGE SPECIFICATIONS

There are package specifications that contain many type or subtype declarations. Out of a total of 153 package specifications, there are 93 package specifications with not more than one type declaration, resulting in a score of 61%.

The presence of multiple types in a package may indicate that conceptually different objects are being defined in a single package. The need to reference any of these objects results in the ability to access all of these objects even when such access is not desired.

EXAMPLE: In X25_data

```
package ACP5250_Types is
--$ . . .
type System_Control_Record_Type
   (Control : ACP5250_Parameters_Type  : = Link Disable;
    Length  : Octet  := Control_Record_Length (Link_Disable) ) is
    record
--$ . . .
type Host_Command_Type
   (Command  :  Host_supervisor_Command_Type  : =
    Host_System_Control;
    Byte_Count : Octet :=
Host_Supervisor_Length (Host_System-Control);
   Control_Count : Control_Count_Type :=
   Control_Count_Type'first)
is
   record
```

SAMPLE MODIFICATION:

```
package system_control_package is
--$ . . .
type System_Control_Record_Type
   (Control  :  ACP5250_Parameters_Type  := Link_Disable;
    Length  :  Octet  := Control_Record_Length (Link_Disable)  ) is
   record

package host_command_package is
--$ . . .
type Host_Command_Type
   (Command  :  Host-supervisor_Command_Type  :=
   Host_System_Control;
   Byte Count  : Octet :=
Host_Supervisor_Length (Host_System_Control);
   Control_Count  : Control_Count_Type  :=
   Control_Count_Type'first)
is
   record
```

REFERENCE: ADAMAT Reference Manual, Section 3.82, Page 3-85.

FIGURE 12 METRIC EXPLANATION FROM WIS REPORT

AVOID NUMERIC TYPE DECLARATIONS THAT REFERENCE SYSTEM-DEFINED TYPES

Most of the numeric type declarations reference system-defined types. There are 2 numeric type declarations that do not reference system-defined types from a total of 10 numeric type declarations, resulting in a score of 20%.

References to the system-defined types force the compiler to use system dependent types rather than universal types.

EXAMPLE: In WISNAS_Message_Types_

```
type One_Octet_Type     is NEW INTEGER range 0 . . 255;
   for One_Octet_Type'SIZE use 8'
type Two_Octet_Type     is NEW INTEGER range 0 . . 65535;
   for Two_Octet_Type'SIZE use 16;
```

SAMPLE MODIFICATION:

```
type One_Octet_Type     is range 0 . . 255;
   for One_Octet Type'SIZE use 8;
type Two_Octet_Type     is range 0 . . 65535;
   for Two_Octet_Type'SIZE use 16;
```

REFERENCE: ADAMAT Reference Manual, Section 2.27, Page 2.29.

FIGURE 13 METRIC EXPLANATION FROM WIS REPORT

4.2 REPORT OF AUTOMATED ANALYSIS

The report of the automated analysis consists of the 411 generated metrics reports. Each of these reports contains only those metrics with a score "less than 1". The reports were presented in this manner to allow the Air Force and the software contractor to isolate those files containing code having characteristics that unnecessarily detract from quality. Figure 14 shows such a report.

```
GOOD     TOTAL     METRIC NAME
•        •         anomaly management
22       39          user_types
•        •           applicative_declarations
•        •           default_initialization
                     . . .
•        •         independence
•        •           macharithindep
                     . . .
•        •         self_descriptiveness
57       62          no_predefined_words
•        •           number_of_commented_declarations
•        •           number_of_commented_statements
5        6           number_of_commented_bodies
•        •           number_of_commented_specifications
•        •           . . .
•        •         simplicity
•        •           decisions
•        •           structured_branch_constructs
6        72          declarations_contain_literals
•        •           array_range_explicit
•        •           subtype_declarations_explicit
•        •           calls_to_routines
                     . . .
•        •         system_clarity
•        •           qualified_aggregate
•        •           named_aggregate
•        •           named_loops
•        •           single_object_declaration_lists
•        •           for_loops_with_type
•        •           no_while_loops
•        •           expressions_parenthesized
•        •           non_negated_boolean_expressions
```

FIGURE 14 ABBREVIATED METRICS REPORT FOR A SINGLE FILE

5. CONCLUSIONS

Our analysis of Air Force-supplied Ada source indicates that an automated, hierarchical, Ada-specific, software metrics framework is an effective aid in the review of the quality of large segments of Ada code.

This work demonstrates that software metrics are useful for controlling quality as soon as the first specifications become compilable. Based on the reported metric findings, the Air Force was able to direct the software contractor to eliminate code characteristics which were deemed by t.e Air Force to be unnecessarily detracting from quality. The metrics analysis provided ESD with a means of addressing quality concerns in the WIS Ada source before the code became executable and the testing results became available.

Proponents of Ada should be encouraged by the characteristics actually measured in this large segment of Ada software. The library, package, type, and exception mechanisms of Ada are, from a software engineering perspective, being effectively utilized. The analysis indicates that this segment of the WIS software employs the features of the Ada language in a manner consistent with the goals, concepts, and spirit Ada was designed to support.

References

[Keller 85]
Keller, S. E., Perkins, J. A., "Ada Measurement Based on Software Quality Principles", Washington Ada Symposium, March 1985, pp. 195-203.

[Perkins 85]
Keller, S. E., Perkins, J. A., "An Ada Measurement and Analysis Tool", Third Annual National Conference on Ada Technology, March 1985, pp. 188-196.

[Perkins 86]
Perkins, J. A., Lease, D. M., Keller, S. E., "Experience Collecting and Analyzing Automatable Software Quality Metrics for Ada", Fourth Annual National Conference on Ada Technology, March 1986, pp. 67-74.

[Perkins 87]
Perkins, J. A., Gorzela, R. S., "Experience Using an Automated Framework to Improve the Quality of Ada Software", Fifth Annual National Conference on Ada Technology, March 1987, pp. 277-284.

ABOUT THE AUTHORS

John Perkins is a member of the Software Research and Development Group at Dynamics Research Corporation. He has a Bachelor of Science degree in Mathematics from Purdue University and a Master of Science degree in Mathematics from the University of Illinois. He has been involved in the development of translators for multi-processor scientific computers and in the development of an attribute grammar-based translator-writing system. He is currently involved in the definition of a quality metrics framework specific to Ada and in the specification of a rule-based consultant for determining software quality goals.

Jane Anderson is a member of the Software Research and Development Group at Dynamics Research Corporation. She has a Bachelor of Arts degree in Mathematics from Brown University and a Master of Science degree in Mathematics from the University of Lowell. She has written a functional requirements document for a complexity analyzer which analyzes the complexity of Ada source in terms of the constructs. She has also been involved in testing ADAMAT.

EVALUATION OF EXISTING BENCHMARK SUITES FOR ADA

ARVIND GOEL AND ERWIN WONG

TAMSCO CECOM

Abstract: This paper evaluates some of the existing benchmark suites for Ada, namely the Ada Compiler Evaluation Capability (ACEC) test suite, ACM Performance Issues Working Group (PIWG) benchmarks, and the University of Michigan benchmarks. The benchmarks have been analyzed with respect to a) evaluating compilers for for real-time embedded applications, b) accuracy of the measurements, c) preventing unwanted code optimizations, and d) portability of the benchmarks. Areas for which additional benchmarks are needed are identified.

1. Introduction

The principal goal of Ada is to provide a language supporting modern software engineering principles in the design and development of real-time embedded systems software. Unfortunately current Ada implementations don't allow the development of real-time embedded software reliably and without sacrificing quality and productivity. The reasons for this are manifold but the most important ones are a) lack of certain features in Ada language itself (this issue is addressed elsewhere [4],[5],[6]) and b) implementation and size of the Ada Runtime System (RTS) which differs widely from one compiler to another. Real-time embedded systems are characterized by severe timing and memory constraints. In traditional real-time systems not programmed in Ada), a separate executive was responsible for making sure that the various timing and memory constraints were satisfied by different parts of an application program. In Ada, the executive is part of the language as the runtime system.

Real-time programmers have no control on the design and implementation of the RTS except that the RTS satisfy the requirements listed in the Ada Language Reference Manual (LRM). Due to the effect on program efficiency and reliability of the various runtime implementation options, simply adopting a compiler that implements the language as defined in the LRM is insufficient for real-time embedded systems. Benchmarks are needed to determine the performance of various Ada language and runtime features in order to assess a compiler's suitability for real-time embedded system applications.

Developing benchmarks for Ada is different from other languages because of the RTS which implements features such as tasking, memory management, interrupt handling, exception handling and others. The task of determining what is to be measured and how is not straightforward. Incorrectly designed benchmarks used to select an Ada compiler for embedded applications can cause an application to be doomed from the very beginning. Techniques are needed to ensure the accuracy and relevance of benchmarks. It is the authors' view that benchmarks that determine the implementation characteristics of an Ada RTS are needed along with benchmarks that measure time and space utilization [7].

Recent studies that have reported results of running existing benchmarks [1] have been conducted on self-hosted compilers where it is difficult to predict the interference effect from the operating system, paging as well as other sources. In an embedded system, there is no virtual memory paging or system daemons and hence there is no effect of such interference on benchmark results. It has to be emphasized that benchmarks that are designed to measure performance of Ada compilers for real-time embedded applications are intended to run on bare targets if meaningful results are to be obtained.

The authors have been involved with developing benchmarks for Ada language and runtime features considered important for programming real-time embedded applications. As part of this ongoing effort, existing benchmark suites were analyzed to determine their suitability for evaluating Ada compiler systems for embedded applications. The test suites analyzed included the Ada Compiler Evaluation Capability (ACEC) suite [8], ACM Performance Issues Working Group (PIWG) benchmarks, and the University of Michigan benchmarks [1]. This paper presents the results of that analysis. The benchmarks have been analyzed with respect to

- the features that the benchmarks are intended to measure and usefulness of the benchmarks for embedded applications. The next section lists the Ada language and runtime features considered important for programming real-time embedded systems.
- accuracy and repeatability of results;
- techniques used for preventing compiler optimizations;

- and portability of the benchmarks.

The authors intend to run some of the existing as well as the newly developed benchmarks under this effort on bare targets, the results of which will be published later. However, we did run these benchmarks (ACEC, PIWG, and U. of Mich.) on an Ada compiler system hosted and targeted for the MicroVAX II.

2. *Real-time Ada Features*

This section highlights the Ada features that a benchmarking suite intended for evaluating real-time embedded systems should address. Within each feature, there is a brief description of the various aspects of that feature that should be benchmarked. These aspects are related both to runtime performance as well as determining implementation characteristics. Of course, this list is by no means exhaustive and further additions will be made to this list as the work progresses [7].

2.1 Tasking

There is a significant amount of necessary overhead inherent in the programming constructs associated with Ada tasking. Tasking overhead affects the efficiency of the RTS in both sizing and timing as the RTS contains the code that implements the tasking features (entry calls, accepts, selects, .. etc.). The LRM outlines the interface to the tasking system from an applications program and a method of communication and synchronization between tasks, but has left a large part of the implementation undefined.

Some of the aspects of tasking that need to be benchmarked include:

- Time to activate and terminate tasks. Task activation and termination times are measured for task objects of a task type, tasks declared directly, and tasks allocated via the new allocator.
- Determine if task space is deallocated on return from a procedure when a task that has been allocated via the new operator in that procedure terminates.
- Determine the maximum number of tasks that can be elaborated in a system without running out of memory.
- Determine if tasks performing I/O may block an entire process thus defeating task concurrency.
- Determine the status of tasks declared in library packages on termination of the main program.
- Time required for a) simple rendezvous and b) for passing different sizes and types of parameters during rendezvous.
- Default priority of tasks (and of the main program) that have no defined priority.
- Algorithm used when choosing among branches of a selective wait statement.
- Order of evaluation for guard conditions in a selective wait.
- If a low priority task activation could result in a very long suspension of a high priority task.
- Priority of a rendezvous between two tasks without explicit priorities.
- Order of evaluation of task names in an abort statement.
- Determine which tasking optimizations (e.g., Habermann-Nassi) are implemented.

2.2 Scheduling and Delay Statement

Task scheduling is an important consideration for a multitasking application. Real-time embedded systems contain jobs with hard deadlines for their execution. Failure to meet a deadline reduces the value of the job's execution possibly to the extent of jeopardizing the system's mission. It is the responsibility of the RTS's scheduling mechanism to guarantee that the deadlines are met.

Some of the things that we need to know about the scheduling mechanism include:

- Determine if user tasks are pre-emptive. Does a completed delay interrupt the currently executing task to allow the scheduler to select the highest priority task.
- Determine the method of sharing the processor within each priority to prevent the starvation of any single task (round-robin, time-slicing).

2.3 Memory Allocation/Deallocation

Ada is the first high order language intended for mission critical, real-time applications that requires dynamic memory allocation and deallocation. The Ada language encompasses dynamic objects of unconstrained types, objects of access types, workspaces of tasks, compiler generated temporary objects for computation, and subprograms with locally defined data. Dynamic memory allocation and deallocation poses efficiency and reliability concerns in a real-time embedded system environment.

Some of the aspects of memory allocation/deallocation that need to be benchmarked include:

- Time for allocating storage known at compile time.
- Time for allocating variable amount of storage.
- Memory Allocation via the New Allocator
- Determine STORAGE_ERROR threshold.
- Determine if garbage collection is performed.

- Determine if Unchecked_Deallocation is implemented.
- Measure time required for Unchecked_Deallocation.

2.4 *Exceptions*

Real-time embedded systems should be able to handle unexpected errors at run-time. Unexpected errors could have disastrous consequences if not handled properly. Many real-time systems operate for long periods of time in stand alone mode and there is a need for efficient and extensive error-handling for such systems.

Benchmarks for exceptions include:

- Timing overhead due to exceptions. Measure overhead if a code sequence has an exception handler associated with it, yet no exceptions are raised during the execution of that code.
- Measure exception response time. Exception Response time is defined as the time when a exception is raised to the time the execution handler starts executing.
- Measure exception propagation time. Exception propagation time is the time between raising an exception in a unit and the time required to propagate the exception by raising the exception at the point where the unit was invoked. No exception handler is present in the unit where the exception was originally raised.

2.5 *Chapter 13 Benchmarks*

Ada defines some features which allow a programmer to specify the physical representation of an entity, i.e., map the abstract program entity to physical hardware. Real-time embedded systems require Chapter 13 features to interface with physical devices and in specifying the precise layout of data structures. These features are implementation-dependent: an implementation is not required to support these features. For real-time embedded systems, it is necessary to that the Ada LRM Chapter 13 features be implemented and made mandatory.

Benchmarking Chapter 13 features depends on the characteristics listed in package SYSTEM, the hardware and its interface with the peripheral devices. The goal should be to develop general purpose benchmarks that can be easily tailored for a specific implementation.

- Determine if address clause can be specified for an object, subprogram, package, task, or single entry of a task family.
- Determine if the length clause, enumeration representation clause , and record representation clause are implemented.
- Determine the availability of bit packing via pragma PACK for arrays of boolean and measure the overhead required for shift, rotate, and bit-wise boolean operations.

2.6 *Interrupt Handling*

In real-time embedded systems, efficient handling of interrupts is very important. Interrupts are asynchronous events. They are hardware or software signals that stop the current processes of the system under specified conditions and in such a way that the processes can be resumed. In a real-time embedded system, interrupts are critical to the ability of the system to respond to real-time events and perform its required functions and it is essential that the system responds to the interrupt in some fixed amount of time. Benchmarks include:

- Measure Interrupt Response Time.
- Determine if accept statement executes at the priority of the hardware interrupt, and if priority is reduced once a synchronization point is reached following the completion of accept statement.

2.7 *CLOCK Function Overhead, Resolution and Type DURATION*

For real-time embedded systems, the CLOCK function in the package CALENDAR is going to be used extensively. The CLOCK function reads the underlying timer provided by the system and returns the value associated with the timer. If the overhead associated with executing the CLOCK function is high, then real-time embedded systems will be hesitant to use the CLOCK function.

The Ada type DURATION is not required to have the same resolution as the clock period. It is required by the Ada LRM to be at most 20 milliseconds and that it be no more than 50 microseconds. A real-time embedded system has timing constraints that require response within a predetermined time interval. The clock period (or time resolution) or resolution of type DURATION must support these requirements. Another extensive use of the CLOCK function is for the measurement of time in generic benchmarks.

- Measure CLOCK function overhead.
- Measure CLOCK resolution. This test measures the resolution time of the CLOCK function.
- Determine implementation of type DURATION. Implementation of Type DURATION will determine the resolution of the delay expression.

2.8 *Arithmetic For Types TIME And DURATION*

For real-time embedded systems, it is necessary to dynamically compute values of type TIME and DURATION [1]. An example of such a computation is the difference between a call to the CLOCK function and a calculated TIME value. This value may be used as a parameter in the delay statement. If the overhead involved in this computation is significant, the actual delay will be longer and this could be disastrous for real-time systems.

- Measure the overhead associated with a call to and return from the "+" and "-" functions provided in the package CALENDAR.

2.9 Subprogram Overhead

In Ada, subprograms rank high among program units from a system structure point of view. Systems designed and implemented in Ada appear as a collection of packages and subprogram units, each of which may have multiple procedures. For real-time programmers to use good programming techniques and structured system design methodologies, it is important that subprogram call mechanism be as efficient as possible.

- From our own experience as well as after analyzing existing benchmarks, subprogram overhead has to be measured for inter- and intra-packages as well as generic and non-generic instantiations of code. In all the tests, various numbers and types of parameters are passed with modes in, out, and in out. For intra package calls, all the tests have to be executed with pragma INLINE for the called procedure.

3. Criteria for Benchmark Evaluation

The benchmarks have been analyzed with respect to the following characteristics:

1. Features measured by the Benchmarks: The previous section highlighted some of the aspects of Ada features that are important for real-time Ada programming. Although this paper concentrates on runtime benchmarking, other criteria that are important for a compiler selection include compilation speed, development and debugging tools, documentation, compiler/linker options and operation, library management, and configurability and size of the runtime system. The size of the runtime system is an important consideration for embedded systems as memory is at a premium in such systems. The larger the size of the RTS the lesser is the memory available for an application program. Another important criteria is the configurability of the RTS. A compiler system that loads only those features of the RTS that are needed as well as enables a user to configure the RTS to suit his/her application needs is preferred.

   In the analysis section of each benchmark suite, there is a general discussion of the features measured by that suite. The areas not covered by the benchmarks will also be highlighted.

2. Accuracy and repeatability of results: Benchmarks that produce incorrect data can have disastrous consequences for a project that uses those benchmarks to select a compiler system. It is imperative that the feature be isolated and measured correctly with sufficient accuracy.

3. Preventing Code Optimization: Benchmarks should be designed so that compilers cannot distort results by employing optimizing techniques. The test should be designed so that the feature being measured can be isolated and not removed by the compiler by optimizations.

4. Portability: The benchmarks should be portable and executable on any Ada compiler system with the minimum of modifications. There are some benchmarks (like interrupt handling) that are not portable and depend on the hardware being tested.

4. Ada Compiler Evaluation Capability (ACEC)

The ACEC benchmark suite was put together by the Institute for Defense Analyses (IDA). The purpose of ACEC is to provide users with a) an organized suite of compiler performance tests, and b) support software for executing these tests and collecting performance statistics. These tests were collected by the Ada Evaluation and Validation (E&V) team from several sources. There are around 250 tests in this suite that have been available within the public domain since March, 1986.

4.1 Features Measured by ACEC Tests

The ACEC tests are divided into the following categories:

1. Code Efficiency: For the language features measured, a quantitative measure of its space and time cost is obtained. Each test comprises of two files: a test version and control version. The test version contains the feature under evaluation. The control version must have exactly the same execution time and space requirements except for the use of the specific language feature. The memory space usage for an Ada language feature is computed as the space required for the object code of the test version minus that required for the control version.

   The tests measure

   - space and time efficiency of simple loops, for loop, while loop, if statement, GOTO statement, case statements.
   - Timing for Integer, floating point and fixed point arithmetic.
   - Timing for procedure call overhead.
   - Timing for reference to global, and uplevel variables of access and non-access type as well as components of records.
   - The tasking tests determine overhead in context switches between tasks, impact on performance of guards on entry statements and idle tasks, size of passed parameter in entry calls. The tests also make some attempts to

determine the algorithm used in select statements.

- There are some feature tests that measure optional language features. These tests are few in number and are limited to testing the implementation of some pragmas.
- Some commonly used sorting algorithms and programs like Whetstone and Dhrystone are also provided.

2. Capacity Tests: Capacity tests indicate the limitations imposed by the compiler and the RTS on application developers (e.g. levels of recursion, size of stack). For real-time embedded systems, it is useful to have an idea of the maximum number of tasks that can be elaborated. This is dependent on the amount of memory available to the system.
3. Code Optimization tests: The code optimization tests check for optimization techniques implemented by a compiler. The optimization techniques tested for include loop optimizations, common subexpression elimination, expression simplification, strength reduction, constant propagation, boolean, constant, and numeric folding, function call elimination, and others.

It is obvious that the features measured by the ACEC tests are not adequate for selecting a Ada compiler for real-time embedded applications. Recently, there is a new effort by Boeing Aerospace Company to develop a new set of ACEC benchmarks. More information on this effort is lacking at the present time.

4.2 Accuracy

The ACEC tests provide facilities to measure elapsed time as well as cpu time used during execution. The accuracy with with a feature can be measured depends on the SYSTEM.TICK divided by the number of iterations of the benchmark [1]. The control and test loops are executed 10000 times regardless of the clock resolution. If the clock resolution is 10 milliseconds, the results can be measured within an accuracy of 1 microsecond for 10000 iterations. But if the clock resolution is more (say 100 milliseconds), 10000 iterations will not produce results within an accuracy of 1 microsecond.

4.3 Preventing Code Optimizations

The ACEC benchmarks do a poor job of preventing unwanted compiler optimizations. For control and test loops, the benchmarks contain a for loop with a constant iteration limit thus enabling the compiler to perform unwarranted optimizations. The tests for integer, floating and fixed point arithmetic can be easily optimized by a compiler resulting in incorrect results.

4.4 Portability

The CPU clock function for the machine on which the benchmarks are being run has to be entered in one c the packages. The command files needed for the compilation of the tests have to be tailored for the system on which the benchmarks are being compiled.

5. PIWG

The PIWG benchmark suite consist of Ada performance tests that were put together and developed by the Association of Computing Machinery (ACM) Special Interest Group on Ada. There are two versions of the PIWG suite available for distribution. They are PIWG'85 released in December 1985 and an enhanced version PIWG'86 released in August 1986. The PIWG benchmarks comprise a readily available test suite that measures execution times of individual Ada features.

5.1 Features Measured by PIWG

The PIWG tests are divided into three groups.

1. The first group of files establishes the basic routines in the program library. It contains PIWG defined library routines that are needed for the execution of other tests. It also contains some composite benchmarks (Whetstone, Dhrystone etc.).
2. The second group consists of runtime tests that measure the performance efficiency of individual features of the Ada language.
   - Task creation and termination times are measured for task objects of task tvoes and tasks declared directly in main procedure. These tests do not measure task creation and termination times for tasks created via the NEW allocator.
   - Measure the time for dynamic array allocation/deallocation, dynamic record allocation/deallocation.
   - Exception handling and propagation timings.
   - Coding style tests.
   - Subprogram overhead.
   - And task rendezvous times.
3. The third group consists of tests that measure compilation speeds. These are the Z tests.

5.2 Accuracy

The PIWG benchmarks use the dual loop strategy to determine the performance efficiency of an Ada feature. The PIWG tests calculate the number of iterations a benchmark should be run by first determining the minimum duration of the test loop a benchmark should run. The minimum duration is the maximum of 1 second, 100 * SYSTEM.TICK and 100 * DURATION'SMALL. The PIWG tests then calculate the number of iterations a benchmark should be run by starting the iteration count at 1 and increasing it until the test duration is greater than the minimum duration. On a system where SYSTEM.TICK is 1 millisecond, and the desired accuracy is within 1 microsecond, the number of iterations should be 1000. Some Ada features that take microseconds to execute can be measured up to an accuracy within 1 microsecond by running for a specific duration, but features that take longer to execute cannot be measured to within an accuracy of 1 microsecond by this method.

The PIWG tests also output a zero value for tests that produce negative results.

5.3 Preventing Code Optimizations

The PIWG tests do not perform a thorough job of code optimizations. Some cases were discovered where it is possible for a compiler to perform optimizations causing distortions in the results.

The PIWG tests provide a set of tests that can be used for evaluating Ada compiler systems for time-shared use. As far as real-time embedded applications are concerned, these tests do not address a large number of features as listed in Section 2. Also, the features that are addressed are not covered extensively. In their current form and shape, the PIWG benchmarks are not suitable for measuring the performance of Ada language and runtime features important for embedded systems.

6. University of Michigan Benchmarks

The University of Michigan has developed a suite for benchmarking specific Ada language and runtime features that are important for real-time embedded systems [1].

6.1 Features Measured by Univ. of Michigan Benchmarks

The University of Michigan benchmarks is a good start towards developing a test suite for evaluating compilers for real-time embedded applications. The features measured by the benchmarks include:

- Time to activate and terminate tasks (for task objects of a task type, tasks declared directly, and tasks allocated via the new allocator). Time required for simple rendezvous and for passing parameters during rendezvous.
- Determine if user tasks are pre-emptive.
- Time for allocating storage known at compile time, allocating variable amount of storage and memory allocation via the new allocator. Determine if garbage collection is performed, if Unchecked_Deallocation is implemented, and STORAGE_ERROR threshold,
- Measure exception response time and exception propagation time.
- Measure CLOCK function overhead and CLOCK resolution.
- Measure the overhead associated with a call to and return from the "+" and "-" functions provided in the package CALENDAR.
- Measure subprogram overhead for inter- and intra-packages as well as generic and non-generic instantiations of code.

The University of Michigan is the first test suite that has taken a comprehensive look at determining the performance efficiency of Ada features important for real-time embedded systems. This suite addresses a lot of issues that are mentioned in Section 2 and is an excellent first step towards achieving the goal of a set of benchmarks intended for embedded systems.

6.2 Accuracy

As mentioned above the accuracy with which a feature can be measured depends on SYSTEM.TICK divided by the number of iterations of the benchmark. Most of the Michigan tests have a iteration count of 10000, although this number can be changed by the user to produce the desired accuracy.

The dual loop benchmarks make the assumption that the overhead in calling the CLOCK function is fixed and hence this gets filtered out when the test loop and control loop timings are subtracted. This assumption may not be true and for each system that has to be benchmarked this assumption has to be verified.

In a recent report published by the Software Engineering Institute [2], and also from the experience the author had in running the University of Michigan benchmarks some negative results were encountered in running these benchmarks. If the control loop is a subset of the test loop, then the timing difference between the test loop and the control loop has to be positive. But due to factors such as placement of code into memory and asymmetrical translation (where the sequence has fewer machine code instructions) it is possible to get negative results [2],[3]. Hence, before dual loop benchmarks are run on a system, it is necessary to verify that the loop times are similar by coding identical loops in a procedure and comparing their execution times.

6.3 Preventing Code Optimizations

In dual loop benchmarks it is necessary to employ techniques that thwart optimizations by a compiler. This can be done by hiding constant variables from view, preventing simplification of loop constructs, and by arranging the order of compilation for similar purposes.

The following paragraph has been stated as is from the University of Michigan report [1]. "The key to avoiding code optimization is to not let the compiler see constants or expressions in the loops whose times are being measured. For example, instead of using a for loop with a constant iteration limit, a while loop is used with the termination condition being the equality of the index variable to an iteration variable. The index variable is incremented by a procedure, the body of which is defined in the body of a separate package. The iteration variables are declared and initialized in the specification of a library package. Since the iteration values are kept in variables (not constants), and the body of the increment procedure is hidden in the body of the package, there is no way the benchmark loops can be removed by optimization as long as the package specification and body are compiled separately with the body being compiled after the benchmarking unit. Similarly, the compiler must be prevented from removing the execution of the feature being tested from the loop or eliminating the loop entirely from the control loop which does not contain the feature. To ensure that these problems do not happen, control functions are inserted into both loops and the feature being measured is placed in a subprogram called from a library unit. Again, if the bodies of these subprograms are compiled separately, and after the benchmark itself, there is no way for a compiler to determine enough information to perform optimization and remove anything from the control or test loops."

6.4 Portability

The Michigan benchmarks enable the user to specify the number of iterations the control and test loops can be executed. As discussed before, this number depends on the accuracy desired. For certain tests like task activation/termination, the number of iterations may have to be reduced (thus reducing the accuracy of the results).

7. Conclusions

Benchmarking Ada implementations to determine their suitability for real-time embedded systems is an extremely complex task. This job is made even more difficult due to differing requirements of various real-time applications. The University of Michigan benchmarks are a good start towards developing a comprehensive set of benchmarks for embedded applications. In the near future, the authors plan to develop benchmarks to address the issues raised in this paper as well as run those benchmarks on embedded system compilers.

REFERENCES

[1] R.M. Clapp et al., "Towards Real-time Performance Benchmarks for Ada", CACM, Vol. 29, No. 8, August 1986.

[2] N. Altman, "Factors Causing Unexpected Variations in Ada Benchmarks", Technical Report, CMU/SEI-87-TR-22, October 1987.

[3] N. Altman et al., "Timing Variation in Dual Loop Benchmarks" , Technical Report, CMU/SEI-87-TR-21, October 1987.

[4] "Catalogue of Ada Runtime Implementation Dependencies", ARTEWG Report, November, 1986.

[5] "Catalogue of Interface Features and Options for the Ada Run Time Environment", ARTEWG Report, October, 1986.

[6] "Technology Insertion For Real-time Embedded Systems", Labtek Inc., July, 1986.

[7] A. Goel, "Real-time Performance Benchmarks For Ada", TAMSCO Technical Report, December, 1987.

[8] "User's Manual For the Prototype Ada Compiler Evaluation Capability (ACEC)", Institute For Defense Analysis, October, 1985.

About the Authors:

ARVIND GOEL

ERWIN WONG

Arvind Goel received his B. Tech. degree in Electrical Engineering from IIT, Kanpur, India in 1980 and MS degree in Computer Sciences from the University of Delaware in 1982. Presently, he is a computer scientist at Technical and Management Services Corporation where he is leading an effort to develop benchmarks for evaluation of Ada compilers intended for real-time embedded applications. His interests include programming languages, Ada compiler evaluation, APSE research and evaluation, and developing software for embedded applications and distributed targets.

Mailing Address:
TAMSCO, 145 Wyckoff Road
Eatontown, NJ 07724

Erwin Wong is an electronics engineer with the Center for Software Engineering, US Army CECOM. He currently oversees the evaluation and development of Ada compiler performance tests. His research interests include compiler development, programming languages, and programming support environments. He holds MS degrees in Computer Science and Electrical Engineering from Fairleigh Dickinson University.

Mailing Address:
HQ US Army CECOM
Center for Software Engineering
Attn: AMSEL-RD-SE-AST-SS-A (E. Wong)
Ft. Monmouth, NJ 07703

INDUSTRY USE OF A MULTI-LANGUAGE SOFTWARE DEVELOPMENT ENVIRONMENT

BEVERLY J. SCHULTZ

DIGITAL EQUIPMENT CORPORATION

ABSTRACT:

For software developers world-wide, there has been a need to create a usable software development environment which includes more than one language. This need has been felt strongly at Digital Equipment Corporation. Digital's response to that need is the substance of this paper. Specifically, this paper addresses how Digital's environment has evolved to its present state, what the present software development environment includes, how Digital developers use it, how the integration and versatility of this environment make it viable for large-scale applications, and some general directions that Digital has for the future of its software development environment.

1 INTRODUCTION

Engineering software development environments for interactive software construction has been completed in varying degrees by many vendors and software companies. In addition to providing an adequate environment for customers, vendors like Digital Equipment Corporation [1] need strong software development environments internally to produce software required to support their hardware and to make software development a profitable business.

One of the exciting developments during the last eight to ten years of business in software at Digital is the evolution of a very strong software development environment to meet the needs of internal users as well as external customers. The extension of this is that we have learned lessons from our efforts and these have influenced our further work as we continue to develop languages and tools for a fully integrated software development environment.

This paper will address four topics:

1. The evolution of Digital's environment to its present state
2. The current software development environment on VAX/VMS and how Digital developers use it
3. The integration and versatility of languages and tools to create a viable environment for use in large-scale applications
4. Directions for the future

2 EARLY DECISIONS IN AN EVOLVING ENVIRONMENT

2.1 History of Digital Compilers

In the late 1960's, software developers were primarily interested in using strong compilers. In response, Digital concentrated in producing compilers that were industrial-strength. Digital built compilers for its PDP-11 series machines, and in the process learned that as good as our individual compilers were, there was a strong need for a standard architecture for passing information in the "environment" that a developer was using. When a new operating system was developed, VAX/VMS, much was done to make the operating system itself easy for use in software development. Moreover, instead of putting many compilers on this new operating system in a random way, a great deal of thought went into creating a standard architecture for the family of compilers that would reside on VAX/VMS. Creating a common calling standard and common passing mechanisms, and including a standard condition handling facility were positive steps toward calling any language from any other. This helped promote reusable code and consistent methods of development, and it also meant that tools and utilities created on VAX/VMS could be multi-lingual. We profited from the common calling standard, and internally have many products written in more than one language, even though we have used the BLISS language for the majority of our software development in the last 10 years.

[1] The following are trademarks of Digital Equipment Corporation: VAX. VMS. DEC. DEC/CMS. DEC/MMS

2.2 History of Digital Tools

2.2.1 Debugger with Compilers

The ability to call any language from any other, inevitably meant that the "environment" had to evolve to support multiple languages. Multiple-language support has many advantages, but it increses the complexity of software development for all of the related tools. In developing an Ada Programming Support Environment (APSE), when finished that APSE must work equally well as a Fortran Programming Support Environment or a Pascal or Basic or C Programming Support Environment. For example, the VAX Source Code Analyzer (SCA), when being queried for all of the COMMON BLOCKS in code basically written in Fortran, must also show the related PSECTS for those BLISS modules in the system. SCA must understand that different compilers have different names for similar things, and must work generically across the multiple language modules. Tools must be generic while still being efficient. The VAX Symbolic Debugger uses a symbol table specification which all languages support but still gives the debugger the ability to respond rapidly to the specific needs of individual languages [1]. The advantage of a multi-language environment concept was that people used the same tools on VAX/VMS to meet diverse needs. Users did not have to learn to use a different debugger or editor when assigned to program with a language other than the language they are fluent in. The challenges continued for the VAX Symbolic Debugger as new languages such as VAX Ada (r) were developed at Digital. When VAX Ada was developed, the Debugger had to handle things such as tasking. It would have been easier to develop a specific debugger for Ada, but this would have been impractical for the user. Designing a debugger to work for 12 languages, and to insure the basic needs of each language was met, was a challenge for the Debugger project. More important, for the user trying to handle multi-language systems, such a set of debuggers would have been impractical. Incorporating the calling standards and working diligently to produce a multi-language Debugger produced a very powerful product!

Except in cases like the VAX APL interpreter, which has its own environment, Digital pursues the goal of a common environment for users, regardless of which tools they choose to manipulate on VMS.

2.2.2 CMS and the Debugger with Compilers

Compilers continue to be enhanced at Digital to provide new and interesting features. These are still undergoing modification to meet evolving standards, new hardware, and better technology. Strong compilers on VAX/VMS, plus a strong Debugger, were necessary but not sufficient for software development being done within Digital. Internal developers found that they needed a variety of other tools to help them complete their projects within time constraints with the desired quality.

In 1978, for example, internal developers needed a way to keep track of multiple versions of a code module, so an automated method of code management was designed for internal use. It was enhanced as systems became larger and multiple releases were being made of individual software products. By 1981, this code management system was being used by over one thousand internal Digital developers to help automate and control the complexity of managing their sources. This internal product later was offered externally as VAX DEC/CMS, DEC's Code Management System [3]. CMS can handle any ASCII text file, and the language that the code is written in does not matter.

How was it decided to make it an external product that customers could use in their own development environments? First, the internal tool that had grown to handle many diverse needs was reworked. Its command line was modified to fit the standard command language on the system and to be consistent with other products that were available. It was made much more robust. Error recovery was enhanced, and CMS added the ability to roll back any command that was "stopped" before completion. Much more testing was added to insure the integrity of its libraries, even though internal people had never lost code in over 4 years. The product was compared with other projects that were being worked on for consistency and for cleanness. CMS was finally released as the VAX Code Management System (DEC/CMS).

Tuning a good internal tool so that it could be used by customers in their own robust environments was doable. While this work was going on, other functionality was needed for which no tool was available. These tools had to be built, and often while they were being built from scratch to satisfy the needs of internal software developers, marketing was putting pressure on the team to already make them external products.

More and better tools were also not enough. It became obvious that they had to integrate well with each other to take advantage of their power easily by the user. But integration proved to be complex to manage and often difficult to do.

For example, several projects began to work together to handle one of the most delicate and troublesome problems - that of a common data base for holding private information. Each product that was evolving needed some kind of control files to hold data that they needed to do their job. If these were stored as simple files on the VMS system, the overhead of opening and closing multiple files was inhibiting. As project teams tried to work together to evolve solutions to the data storage and retrieval problems, they found that what they wanted to store differed from one project to another. Projects found that creation of their own project-specific data bases seemed easier than defining and building data bases common to even two products. Digital had some of the best overall data base products available at the time, but prototyping databases using DBMS and other commercially available products showed that the power of these products was more than what was needed for small applications such as tools wanted, and carrying around a large database for such a small job was prohibited by cost and performance. Strong databases were robust for large applications and for transaction processing, but the small amount of storage and retrieval needed by a simple tool meant minimizing the overhead needed to handle their data. Customers developing software wanted very fast software development tools, and weren't willing to accept a wait longer than they were currently required to complete a file copy command.

Data storage problems were resolved in the short term by allowing each tool to handle it's own manipulation of private data. But when CMS V2 produced a very simple data base to handle data storage for its needs, that mechanism of storage was seen as useful and adapted by other projects to handle private data. Common data held by one tool but needed by another was accessed through callable interfaces. Callable interfaces and standard definition of these interfaces became important, and was a primary means of sharing data and information.

Reusability of code was common within Digital, but evolved in an informal way. Developers were accustomed to using to the VAX Run-Time Library. If one team built a module that others would use, they submitted their routines to the Run-Time Library. Over time, a Tools Clearinghouse evolved to accept that developers created for one job and were willing to share with others. Developers who need a tool look there first. Those who are about to implement something are very apt to first consider if they can take advantage of the work of others, and share code if that seems useful.

2.2.3 Specific Tools Added to the VMS Environment

Environment-enhancing tools that are well integrated evolved quickly. Internal software development became more complex. Customers had a variety of specific needs, and products like VAX FMS (forms management), the VAX DEC/SHELL (a UNIX [1] Bourne Shell), Datatrieve (information management tool) , and VAX DEC/MMS (automated application system builder) evolved to meet these needs. In many cases, what new features were doing was automating software development by letting the computer do things that users had done by hand or that users had been required to keep in their heads.

As the offerings of VAX/VMS products, including its "layered products", continued to expand, Digital worked hard to insure that the combination represented a consistent and powerful environment for software developers. The wealth of third party applications available for users allowed Digital to concentrate on core tools in software development.

2.2.4 More Tools Evolve from New Product Needs

The development of Ada spurred Digital to reassess the direction in which internal software development was moving. With plans for building an Ada compiler had to come plans for an Ada Programming Support Environment that could evolve from the Stoneman requirements of the Department of Defense. Instead of addressing those requirements for Ada alone, the larger challenge was addressed. When the first version of VAX Ada appeared, users had, not a new Ada debugger, but the familiar VAX Symbolic Debugger enhanced to provide full Ada support [2].

The VAX Language-Sensitive Editor was built as *the premier editing environment for Ada users*, even though it was just as powerful and useful for users of Basic, Bliss, C, Cobol, Fortran, Pascal, and PL/I. The real power of an integrated, multi-language environment is apparent to current VMS developers at Digital who are now moving toward doing some, but not all, of their development in Ada. Projects building something like servers may want to use Ada tasking, so they develop those modules in Ada. Their environment hasn't changed. It has just been extended.

3 THE CURRENT SOFTWARE DEVELOPMENT ENVIRONMENT

3.1 Overview

Inside Digital, processing speed on VAX/VMS is important for developing the software products expected of a vendor. But the effective use of that computer power is paramount for both Digital developers and customers. The VAX/VMS environment, as it continues to evolve, isn't a prototype environment used in research, but it is a real working environment for very large scale applications development. Customers of Digital depend on this development environment because it gives them a powerful edge in producing software efficiently. The core of the components of this environment include:

```
VAX/VMS SOFTWARE DEVELOPMENT ENVIRONMENT:

 o VMS SYSTEM
   - System Services
   - Common Run Time Library
   - DCL
   - Special Purpose Libraries
 o BUNDLED TOOLS
   - VAX Symbolic DEBUGGER
   - VAX TPU (Text Processing Utility)
   - DSR (RUNOFF)
   - MAIL
   - SORT
 o UNBUNDLED TOOLS
   - LANGUAGES (ADA,BASIC, BLISS, C, COBOL,
     Fortran, LISP, PASCAL, PL/I...)
   - VAX Language-Sensitive Editor
   - VAX Source Code Analyzer
   - PROJECT CONTROL (DEC/CMS,
     DEC/MMS,...)
   - TEST DEVELOPMENT (DEC/
     Test Manager)
   - Software Performance Analysis
     (VAX Performance & Coverage Analyzer)
   - ALTERNATE COMMAND LANGUAGE
     and UNIX utilities (VAX DEC/Shell)
2
   - Text processing language (VAX SCAN)
   - DATABASES (RDB, DBMS...)
   - Common Data Dictionary (CDD)
   - CONFERENCING (VAX Notes)
   - 4th Generation Languages (DATATRIEVE,
     VAX Cobol Generator)
```

[1] UNIX is a trademark of Bell Laboratories

These features constitute the components of a developer's core software development environment. They produce a foundation from which developers can concentrate on the pieces of software unique to their intended application.

The functionality of the products listed above is enhanced because all of the tools are presented in a consistent way to a developer. The general VAX/VMS interface, called the Digital Command Language (DCL) provides a command language interface used by all of the tools on VMS. Picking up a new tool is easier because it conforms to the style of VMS.

Most tools in the environment are flexible enough to serve multiple functions. Numerous users have used DATATRIEVE (information management tool) and DEC's Code Management System (CMS) to create a powerful configuration management tool for their program. Others use DEC's Module Management System to control the building of documents. CMS provides an audit trail of all that has happened on a project which is used in some companies for analysis of project information and for metrics reporting.

It has been difficult to continue to develop new products and still conform to the multi-language foundation of the Digital tools architecture. However, it is interesting that this concept is enforced internally at Digital because the VMS operating system as well as many of the layered products on VAX/VMS are written in more than one language. Tools as developed are needed to work within a common environment, or Digital developers can't be as productive. Digital developers design software tools that customers will use. Since colleagues will be using them too, they had better be easy to pick up and they needs to fit well with what developers already have.

3.2 Examples

Examples of some of the more recent extensions to the core VAX/VMS environment show the power that comes to the user by means of current technology.

3.2.1 The VAX Language-Sensitive Editor

The Language-Sensitive Editor (LSE) is a multi-language advanced text editor enhanced to lets users quickly and accurately development programs. LSE allows software engineers to code in VAX Ada, VAX Basic, VAX BLISS, VAX C, VAX COBOL, VAX Fortran, VAX LISP, VAX Pascal, and VAX PL/I with the aid of language-specific templates, expansion of language constructs, language-specific help for any construct, and extensive review of all code that does not compile correctly. For VAX C and VAX Ada, LSE makes building a correctly compilable program even easier because LSE provides error correction for their compilation errors.

Using LSE has changed some of the ways in which software was developed at Digital. Developers no longer use hard copy listings; listings are seldom seen in offices. Working on-line became the norm. Developers began to spend less time getting clean compilations and to focus on building good code. Particularly important has been the ability to interactively review errors and the ability to compile, read mail, and work with multiple buffers without exiting the editor.

Internal users at Digital, like many customers, want to tailor their editing environment in their own way to "feel right for them". This need to modify the editor caused LSE to be built on the VAX Text Processing Utility (which comes with VMS) and thus allowed LSE to be completely tailorable as well as programmable. With LSE, developers can modify their editing environment to suit their own desires and needs, a flexibility essential for our programmers.

3.2.2 The VAX Source Code Analyzer

The VAX Source Code Analyzer gave developers an additional edge in developing and modifying source code. The ability to interractively cross-reference ALL of the modules in his system through the VAX Source Code Analyzer (SCA) lets the developer insure the continuity of his work across many files. If a developer wants to change the parameters of a routine while in LSE, a simple query (done while in LSE but handled by SCA) can show him all of the places that routine was referenced. The Editor can then take him to the corresponding source modules at the location to be modified. This can be done for a few references or for many without the user losing the context of his problem.

If a developer, sitting at his terminal and looking at a code module, sees a call referenced on his screen, he can point to that routine, and then go to the primary declaration of that routine by hitting a single keystroke. The primary declaration is displayed in another window quickly and effortlessly. This gives the developer a strong understanding, not only of the module to fix, but of how his fix is going to affect the rest of the code. The developer can insure that newly-added changes won't introduce errors in other modules. Static analysis of code using call-trees with SCA and checking for references that were read but never written (or written but never read!) tightens the code and reduces chances for error.

Digital developers started using an early prototype of SCA in 1985, getting used to looking at their source code has a system and not as individual modules. Developers using SCA quickly adjust to moving around their system asking questions like "find all of the places this routine was used" or "find all of the symbols that start with 's'". They easily "gather data" about the implications of a change to code, and are confident that their modifications have the least impact (or the best payback!) on the rest of the system. The power of LSE and SCA together gives users on-line development at its best. Because the prototype was in use while the product was being developed, extensive internal user feedback was gained to modify the final product for optimum usability and needed features.

[2] UNIX is a trademark of Bell Laboratories.

3.2.3 VAX DEC/CMS—Code Management System

To manage change, developers use the Code Management System (CMS) libraries for their code storage. They find that they have a full audit trail of all of the changes that have occurred over the full phase of development. They can be fixing bugs on earlier released versions of the code while they are currently in new development. The disk storage savings using CMS libraries is substantial. DEC/CMS can be used for documents or memos or any kind of VMS files but it has been optimized for source code and as become essential to project development. Creating and "freezing" baselines is straightforward with CMS. Being able to go back and retrieve modules from an old baseline while continuing with new development is essential.

Some of our projects are developing at least three variant products with a common set of sources, and use CMS to make order out of this chaos. Our VMS documentation group has used CMS for years to keep multiple versions of all of the VMS documentation. Hardware groups have used CMS over the years to store chip layouts. Thousands of Digital developers store their source code in CMS and use it to handle complex development. Groups like VMS have put a layer on top of CMS to handle the complexity of their large scale application.

3.2.4 The VAX Symbolic Debugger

The VAX Symbolic Debugger is an integral part of the VMS environment during implementation. The Debugger allows state-of-the-art debugging with multiple windows. Debugger features let the developer to walk through his code and clearly see what is causing problems. As with all the VMS tools that work together in a common language and tool environment, the Debugger works across languages. While in the Debugger, you can call the Language-Sensitive Editor and make changes to your source code as you continue your debugging session. You can have access to the VAX Source Code Analyzer information while editing, and include access to files from DEC/CMS in the process.

Junior programmers typically use the Debugger in a limited way and gradually discover its full power. Internal users find the multiple windows of their debugger a particular advantage in debugging programs.

3.2.5 VAX DEC/MMS—Module Management System

The Code Management System does not put the application together or perform consistency checks on the built system, but a developer can use Digital's Module Management System (MMS) for this. MMS keeps all of the developers working without version skew on the same "system." A team of developers can store the results of their individual efforts in CMS libraries on the VMS system, and MMS can get to them directly. MMS omits unnecessary operations and allows for a formal definition of the structure of the system being built. MMS understands VMS files and VMS libraries.

Thus, MMS can build a system efficiently using a minimum of CPU time and with consistent reliability. The dependency file which the developer writes to describe the system to MMS can be stored by baseline in the CMS library along with the code for that baseline. With this method, developers always know what went with a baseline or a release and exactly how that system was built.

Eight hours used to be the norm for an overnight build of a Digital software product. With MMS, product builds complete in as little as 10 minutes with eight hours being the "worst case", and happening only with a total rebuild. MMS is used widely for building documents at Digital as well as executable images.

3.2.6 The VAX DEC/Test Manager

The VAX DEC/Test Manager (DTM) lets a project team automate regression testing. Many developers working simultaneously on a software project typically finish specific coding tasks at different times. They always test their tasks individually, but if they are using the DEC/Test Manager to control their tests, they can simply give their groups of tests to the DEC/Test Manager and have the Test Manager run these groups of tests for them. Before they consider their task finished, they may run their tests for that task with one or two other groups of tests that exercise functionality that the task interracts with. Finding bugs and integrating their software becomes part of the use of the environment when using the DEC/Test Manager.

Most projects use the DEC/Test Manager for Integration Testing, for insuring product integrity, and for checking specific fixes. We have made strong use of DTM as our products have become interactive and screens have had to be tested. The need for bit-mapped graphics testing internally has had DTM developers exploring this area for the past several years.

Our Software Quality Measurement group uses DTM to hold regression tests of all layered products on VMS. They can then run them with each new baseline of the VMS operationg system to insure that new features in VMS don't cause massive rework in the layered products. A change can be withdrawn before it affects large numbers of people and causes large delays and disruptions, since DTM's information can be fed back to the operating system in time to make corrections.

3.2.7 The VAX Performance and Coverage Analyzer

The VAX Performance and Coverage Analyzer (PCA) is a tool used to analyze the run-time behavior of application programs. It measures test coverage and to insure that all paths through the software have been tested. It also finds places that are bottlenecks in performance so developers can concentrate on fixes that will have the biggest payback instead of trying to optimize every part of the code. The VAX Performance and Coverage Analyzer can collect page fault data, data on system service calls, I/O data, exact execution counts, and program counter sampling data. It processes the data for viewing in tabular displays and histograms.

PCA has been used by most Digital software projects, but for different reasons and to achieve different results. We have achieved large gains in performance by spending a small amount of time with PCA, and thereby causing developers to focus their time when such time is limited. Some projects use it regularly just to monitor some aspect of performance. Some use it consistently for test coverage. PCA came into being in the early 1980's because Digital needed something to test its Fortran compiler for performance analysis. From that early tool, PCA evolved into the product it is today.

3.2.8 Other Tools

Digital has added a variety of other tools to its environment in the past two years. The VAX Cobol Generator gives the user the power of a graphical fourth-generation language. This tool converts graphically-expressed program designs into working Cobol code for execution. The VAX Software Project Manager helps project leaders to successfully manage the process of project completion, scheduling, and tracking tasks. Other products support endeavors such as system management, office needs, and document building. All can be used by the software developer on VMS as needed.

4 INTEGRATION AND VERSATILITY OF LANGUAGES AND TOOLS

With the above products as well as other languages and tools available on VAX/VMS, the VMS environment gives an ideal climate for serious software development. The main advantages of using the VAX/VMS environment for software development are integration and versatility.

4.1 Integration of Tools

The complexity of large software projects requires that integrated tools work effortlessly together. This has been the area of greatest technical challenge in *software engineering*. When a developer needs a tool, or a features, it must be there for him to use when he needs it. This is why the compilers generate code that is callable from other languages. If a Fortran program can call a 'C' subroutine and thus take advantage of existing code, work can move more rapidly.

Compilers must be closely integrated with the VAX Symbolic Debugger as well. With the same debugger used for FORTRAN or C or COBOL or BASIC, developers get used to its features and can expect them also for Ada or PL/I, with extensions to handle such things as tasking in Ada.

Integration of tools was particularly necessary in the testing area. One would not want to leave the DEC/Test Manager in the middle of reviewing the results of a large set of tests just to call up the Performance and Coverage Analyzer to determine the test coverage of that set of tests. By calling PCA from inside of the Test Manager when PCA is needed, the user has common capabilities of both tools at the same time.

Integration in the VAX Language-Sensitive Editor (LSE) went one step further. Developers can keep the same editor session going for a long time, editing multiple files and multiple buffers, compiling and reviewing compilations and making many modifications. This required that the Editor be built with very strong compiler support and that multitool accessibility be seamless to the user. LSE is tightly coupled with SCA and with CMS to insure that cross-referencing can be done within the editor with little effort. If a piece of source code has to be pulled from CMS to make modifications it is done without leaving the editor and without the developer having to explicitly call for the file. SCA works tightly with major compilers on VMS to insure that its data matches the source. The user doesn't have to contend with the logistics that puts all of this power at his disposal. The functionality is available as needed, whatever tool is being used.

Integration thus gives the user the sense of one environment, even though he is making use of multiple tools and compilers. The Language-Sensitive Editor can be called from inside the Debugger or from inside PCA and lets you modify your source code as you continue your debugging session. From there you can use the Source Code Analyzer to determine the implications of your changes on the rest of your code. You can make those changes, and then return to the Debugger for your next breakpoint or watchpoint. If you want to, you can run some tests with DTM to insure that your changes are good, and then use the VAX Performance and Coverage Analyzer with those DTM tests so that you can get performance data as well as regression test results from your testing. You can do all this in a single session and be assured that you still have your software's good performance.

Integration also means that third party tools as well as Digital tools can interface with each other. As tools need protocols to communicate with each other, Digital evolves standards to progress toward a truly common tool environment. For example, CMS is now callable from 38 entry points, which helps third-party developers as well as LSE. LSE has a standard diagnostic file format, and external users with a Jovial compiler or some other non-Digital compiler can use that format to build the support that they need for their non-Digital products. LSE is also callable, and can therefore be used from other third party tools. As they become stable tool interfaces and formats are made available to all users to modify this environment for the highest user payback.

Choosing to use multiple tools on VAX/VMS implies that the user has much more power using these tools in conjunction with each other than they would if all were separate entities. In addition, all of the tools use the power of the VMS base. The whole is greater than the sum of its parts.

4.2 Versatility of VAX/VMS Environment

As developers get used to using VMS and its many layered products, we find that the run-time library routines, the standard operating system calls and the library calls all become means for our developers to use 'reusable code' and to focus more attention on unique applications' problems. We have developed products for use on PDP/11s, for TOPS-20 machines, and for other systems on VAX/VMS because it makes software development faster. "Faster" means more reliably, of better quality, and usually with more functionality. It is worth our while to provide software engineers with VMS systems loaded with as many layered products as are useful to them on their primary development work stations.

5 DIRECTIONS FOR THE FUTURE

Digital's software development goals include continued enhancement of the current environment to enable consistent software development throughout the life cycle of a project. We are pleased with our current offering of software products because most of them can be used across the lifecycle. We can't change our environment radically because of the great relearning cost to our internal users as well as our external users. Given the complexity of software development both internally and externally, our challenge is to continue to evolve our current environment into one that gives even more automated support to our own people and to our customers. The current topics being explored include: change control and consistency management, formalisms for integrating phases of the life cycle; and software information data bases. Rapid prototyping methods, requirements specification, design tools, and automated configuration management are also being addressed in the context of the existing working environment.

The current VAX/VMS system provides a software development environment that is adequate for large-scale software development. The directions that can be taken to make this an even more robust and automated environment are many, and we are working hard to see what makes our people more productive. Some areas are in early stages of research and their worth has yet to be determined. Enhancements in some areas are already being used internally at Digital. As layers continue to be added to the VMS environment, it is significant that each new emerging tool has more ties to the rest of the current base. A more comprehensive environment will continue to evolve, but the core exists today for developers to do large scale applications programming across languages with current environment tools.

REFERENCES

1. B. Beander, "VAX DEBUG: An Interactive, Symbolic, Multilingual Debugger", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, SIGPLAN Notices, Vol. 18, no. 8 (August, 1983): 173-179.

2. C. Mitchell, "Engineering VAX Ada for a Multi-Language Programming Environment," Proceedings off the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices, vol. 22, no. 1 (January, 1987): 49-58.

3. B. Schultz, "DEC/CMS—A Manager's View", Proceedings of the Digital Equipment Computer User's Society, vol. 8, no. 4 (Spring, 1982): 1347-1352.

Beverly J. Schultz
Digital Equipment Corp.
110 Spit Brook Road
Nashua, NH 03062

Beverly Schultz is a Development Manager with Digital Equipment Corporation. She has been involved with Digital's Languages and Tools group since 1981. She currently manages VAXset products as well as compilers, the VAX Debugger, and tools which make up the evolving APSE environment on VAX/VMS. Beverly has been active in the fields of mathematics and computer science since 1965, and holds an M.S. Degree in Computer Science from the University of Dayton.

A Marriage of Convenience: Developing a Practical APSE for Use with Ada® and DOD-STD-2167

Damon Lease
Strategic Electronic Defense Division
GTE Government Systems Corporation

INTRODUCTION

There has been much controversy since the introduction of the Ada[1] language and the Department of Defense Software Development Standard (DOD-STD-2167)[2] to the defense software engineering community. The biggest problem stems from apparent incompatibilities between the newer methodologies espoused by the Ada community and the standard waterfall life cycle and functional decomposition method embraced by DOD-STD-2167. Many of these incompatibilities are now being addressed by the defense software community, but no clear consensus has yet evolved. Defense contractors currently developing Ada software must decide exactly how Ada should be used with DOD-STD-2167 to properly benefit from their respective advantages.

By late 1984, these problems were quite evident at GTE's Strategic Electronic Defense Division. There was no clear division-wide philosophy on the proper use of Ada, and no automated method to support and enforce any methodology that might emerge. Also complicating matters was the pending release of DOD-STD-2167. When GTE was awarded a contract to develop Ada software using a draft version of DOD-STD-2167 as a guideline, the division decided to develop a near-term Ada Programming Support Environment (APSE) for the project. It would support the use of DOD-STD-2167 and generate the required documentation. The resulting environment, now known simply as PSE, has since been adopted for use throughout the entire division.

The primary goals in developing PSE were to:

1. Provide support for Ada software development, incorporating methods used successfully on other GTE projects.
2. Incorporate DOD-STD-2167, tailoring it to the needs of division projects and the proper use of Ada.
3. Automate document generation and other tedious clerical duties, freeing software developers to concentrate on the engineering aspects of the project, thus increasing productivity.
4. Provide support for all phases of the software cycle, not just code generation, and provide support for non-software project personnel.
5. Encourage and teach modern software engineering principles.
6. Incorporate commercial off-the-shelf (COTS) software whenever reasonable to develop an effective APSE as quickly as possible.
7. Reduce the necessary learning time when moving personnel from one project to another.

Conflicts between Ada, contemporary software development methodologies, and DOD-STD-2167 were resolved to provide an interim, flexible methodology, capable of graceful evolution. However, this resulted in some compromises to resolve conflicts.

This paper describes those conflicts and compromises, discusses the philosophy behind the development of PSE, the development methods employed, the current environment's capabilities, including DOD-STD-2167 documentation support, and future directions for the environment.

HISTORY OF PSE

When the decision to implement PSE was made, a few high level management directives were issued to facilitate the development effort. In retrospect, these decisions were a major reason the PSE project has been so successful. The most important decisions were:

- PSE would be an interdisciplinary project, not just a software project.
- High level management would mandate the use of PSE, to guarantee that the tools would be used.
- VAX™ system managers would be included in the PSE design effort, with the intent of optimizing PSE performance.
- The major thrust of PSE would be to enhance the capabilities of the operating system (VMS™) and vendor software products--use them where possible, and not hide them from users.

Therefore, the disciplines of project management, system management, software engineering, and the capabilities of vendors, were intermixed to achieve a well balanced, well thought out design.

It was also decided that PSE would strive to become practical before it became state of the art. This allowed all involved parties to work in an atmosphere where success was quite probable, rather than an unlikely dream.

Lastly, the PSE group was given their own VAX system on which to develop the environment. This allowed the group to use system resources that would have been detrimental to any other development work being done on the same system. This dedicated facility was responsible for the quality and

® Ada is a Registered Trademark of the U.S. Government Ada Joint Program Office

™ VAX and VMS are trademarks of Digital Equipment Corporation rapid availability of some of the more useful PSE tools.

PSE Development Methods

When actual development of PSE started, the first projects scheduled for PSE support were under way. This made it imperative to deploy at least some portions of the environment as soon as possible, ultimately causing a few compromises in the development process.

First, the PSE group decided that PSE would be a set of stand-alone tools, rather than an integrated environment. This made it possible to incrementally release PSE to the supported projects. Second, PSE and the standards and methodologies it was to support were developed simultaneously. Because of this, early PSE tools did not follow the current coding standards; the standards simply did not exist at the time the code was developed. Third, there was no time to generate formal design documentation for each of the tools. Instead, user guides were written for each of the tools, and the actual tool designs were based on the functionality described in the user guides. Fourth, the tools were designed to allow projects to choose their own design methodology. Finally, the group decided that COTS software would be used, when time and cost effective, to increase the speed of the development process.

Other important decisions were made early in the program to optimally support the development and evolution of PSE.

A tool known as DEPARTS (Development Environment Problem and Report Tracking System) was released. DEPARTS is a database and tracking tool used by PSE and its supported projects to record complaints, errors, suggestions, and bugs found within PSE.

Forums were held where PSE users were invited to meet with PSE development personnel to discuss problems with specific tools and suggest enhancements. These meetings continue to this day.

The user guides that served as informal requirements specifications underwent formal reviews. The tool designs underwent in-progress peer reviews. All code was reviewed in code walkthroughs. Also, as tools became available, PSE development personnel became responsible for using the tools and conforming to all standards they enforced.

As PSE development continued, and time and resources became more available, more formalism was introduced. Formal test cases and confidence tests for all tools were written. VAX command procedures were written to install the tool on a given system, ensuring consistency across all systems where PSE resides. Since that time, a concerted effort has been made to rework the earliest tools so they would conform to any standards or methodologies defined after their original implementation.

PSE Development Personnel

Since work began on PSE, approximately seventy-five people have been involved with the project; twenty-four at its largest point. Currently there are twelve people involved with continued maintenance and development.

The project began with six people with software experience, none of whom had Ada development experience. The Design Task Manager (DTM), the Technical Manager (TM), and one additional staff member were experienced software managers. Of the staff later added, several had some development experience, and the balance consisted of new personnel, mostly recent college graduates. As such, the mean experience level of the team was less than one year.

The intent for PSE staffing is to move people between PSE supported projects and the PSE group. There are several reasons for this:

1. Increase the knowledge base of GTE personnel by allowing people to move between projects, thus transferring technology between groups.

2. Prevent the PSE group from becoming an academic group unconcerned with the needs of its users.

3. Provide the PSE group with input from PSE users, and insight into possible improvements in the tools.

4. Provide projects with personnel who know the capabilities and limitations of the tools. These people serve as project specific focal points for PSE help.

DESCRIPTION OF PSE

Currently PSE consists of GTE-developed tools, COTS software, online libraries, a documentation set, and user support and training. The GTE-developed portion of PSE consists of approximately 80,000 lines of Ada code, 20,000 lines of VAX DCL command files, and 2,500 pages of documentation. There are approximately forty tools in the environment. Most of the tools are menu driven and include online help facilities. A complete PSE Software User's Manual exists and the VAX online help facilities have been extensively augmented to include PSE tool descriptions.

The "Stoneman" document[3] defines Ada support environments in terms of a Kernel Ada Programming Support Environment (KAPSE), a Minimal Ada Programming Support Environment (MAPSE), and an Ada Programming Support Environment (APSE). The document lists the requirements for each of these environments:

- A KAPSE "provides database, communication and runtime support functions to enable the execution of an Ada program (including a MAPSE tool) and which presents a machine dependent portability interface."
- A MAPSE "provides a minimal set of tools, written in Ada and supported by the KAPSE, which are both necessary and sufficient for the development and continuing support of Ada programs." The minimal toolset must include a text editor, a pretty printer, a translator, linkers, loaders, a set-use static analyzer, a control flow static analyzer, a dynamic analysis tool, terminal interface routines, a file administrator, a command interpreter, and a configuration manager.
- An APSE is defined as an extension of a MAPSE, extended by increasing the toolset. Suggested additions to the toolset include an Ada program editor, a documentation system, a project control system, a configuration control system, measurement tools, a fault report system, requirements specifications tools, design tools, verification tools, complex translators, and

command interpreters.

The following sections describe how PSE satisfies those requirements.

KAPSE Capabilities

The KAPSE underlying PSE consists primarily of the VAX/VMS operating system, its file storage capabilities, and commercial software products. The database capability is provided by many different tools and structures, including controlled VAX/VMS file structures, relational databases, data dictionaries, and a COTS configuration control tool. File access is controlled through the use of VAX/VMS Access Control Lists (ACLs). Runtime support is provided by the operating system (VMS) and Digital's Ada Compilation System (ACS). ACS provides compilation and linking capabilities. Communications are handled by VMS, via Ada interfaces to COTS software products, and database interfaces.

The GTE KAPSE was assembled almost totally from COTS software because it would not have been cost efficient to develop an operating system, or an in-house relational database package for PSE. These products are readily available, and development costs (money and time) made it more realistic to acquire them from commercial vendors.

MAPSE Capabilities

The MAPSE extension of the KAPSE also consists mainly of COTS software products. It is important to remember that the great majority of tools defined for a MAPSE are also commercially available, and a major emphasis on PSE was the rapid deployment of an effective environment. The GTE-developed tools in the MAPSE are the pretty printer, terminal interface routines, and portions of the command interpreter and configuration manager.

The pretty printer reformats syntactically correct Ada source code according to project formatting standards. Terminal interface routines exist in a configuration controlled VAX/VMS directory. Interfaces are written in Ada, and provide Ada access to standard system-specific interface utilities. The command interpreter is accomplished through the use of VAX command procedures, which provide parameters to Ada programs. The PSE configuration manager is a toolset. The primary tool is the COTS CM tool mentioned above. This tool preserves all data and incremental changes as project configuration personnel approve releases of source files. There are also GTE-developed CM tools including Release Request, Move Plane Request, Do Release, Do Baseline, and Move Plane. Release Request is used by test and development personnel to indicate to CM personnel that source files are ready for release. Move Plane Request is a high level release request tool which "promotes" an entire configuration to its next test or release level. Do Release, Do Baseline, and Move Plane are used by CM personnel to approve and carry out CM requests.

APSE Capabilities

The APSE tools or toolsets mentioned above are addressed individually below. The majority of these tools were developed at GTE.

Ada Program Editor. A syntax directed editor, tailored to GTE coding standards and guidelines is provided.

Documentation System. Two major tools contribute to documentation generation. These tools use the syntax directed editor in conjunction with the Scribe® text formatter to provide templates for generating DOD-STD-2167 documents and other documents used by projects. Scribe allows graphics and text to be merged electronically, so that development personnel may make their own drawings, including them without cutting and pasting, and the drawing files may be put under control like all other source files. Scribe also allows Ada source code or Ada design language descriptions to be included in documents. Thus developers use the same editing and formatting environment for both documents and code. The Document Builder produces tailored templates for the following documents:

- Software Top Level Design Document (STLDD)
- Interface Design Document (IDD)
- Data Base Design Document (DBDD)
- Software User's Manual (SUM)
- Software Programmer's Manual (SPM)
- Software Detailed Design Document (SDDD)
- Unit Test Cases (UTC) Document
- Unit Test Results (UTR) Document
- Integration Test Cases (ITC) Document
- Integration Test Results (ITR) Document
- Software Product Specification (SPS)
- Software Test Plan (STP)
- Software Test Procedures (STPR) Document
- Software Test Report (STR)
- Version Description Document (VDD)

Although PSE is not truly an integrated environment, the Document Builder is an exception. It works in conjunction with other tools and the project database while generating the documentation.

A second tool, the Software Requirements Specification Writer's Environment (SRSW) interfaces with the operating system and the database to aid in generating the Software Requirements Specification (SRS), Interface Requirements Specification (IRS), and System/Segment Specification (SSS) documents.

Project Control System. The Action Item Tracking System (ACTS), the Inventory Tracking System (ITS), the Review Item Disposition Tracking System (RIDS), and the Software

® Scribe is a registered trademark of Scribe Systems, Inc.

Requirements Database (SRD) provide assistance with project control. These tools provide interactive menu driven interfaces to the PSE database, where the status of action items, project inventory, RIDS, and software requirements are stored. SRD performs a major function throughout the lifecycle of a project. It serves as a central repository of data that can be used for preliminary and detailed design, testing, and integration. Managers have full access to these databases, allowing them to rapidly retrieve answers to questions without impacting programmer performance.

Configuration Control System. In addition to the CM tools mentioned previously, the Logical Names tools perform a variety of functions, including some CM functions. The Logical Names toolset performs the following:

1. Sets up standard directory structures for each project, which reflect its design component hierarchy. Within this structure, there are controlled directories belonging to CM. Developers have read-only access to the controlled directories. This structure provides a familiar, standard CM environment, allowing new personnel to quickly understand how project CM works.

2. Creates VAX/VMS search lists to find appropriate versions of files, based on the intent of the user. The search lists are defined by the DEVELOP and CONTROL commands. The developer uses the DEVELOP command to indicate to the system that the most recent version of a file should be found for development work, while the CONTROL command is used by CM personnel to search for the oldest configuration-controlled version of a file.

3. Creates a set of logical names and symbols to quickly access structures with long, cumbersome names.

Measurement Tools. Two COTS software packages provide performance data and coverage data for program execution. One is used primarily as a test tool, and is discussed below with test tools. The other collects sizing and timing data based on execution of the code, providing printed reports. In addition, a software quality metrics tool is currently being evaluated for possible inclusion in PSE.

Fault Report System. There are currently four tools devoted to fault reporting. These tools are DEPARTS, the Trouble Change Report (TCR) tool, the Trouble Report Summary tool, and the Trouble Report Impact Summary tool. DEPARTS, as mentioned above, is used to track bugs and suggested enhancements to PSE. Reports are entered by development personnel as well as PSE customers. TCR tracks Program Trouble Reports (PTR), Software Problem or Change Requests (SPCR), System Trouble Notices (STN), and Hardware Trouble Reports (HTR). A separate TCR database is maintained for each project where PSE is used. Trouble Report Summary provides a detailed report, based on queries, of project PTRs or SPCRs, or a DEPARTS summary. The Trouble Report Impact Summary operates in conjunction with CM tools to determine files that have been or may be affected by a problem resolution or proposed resolution.

Requirements Specifications Tools. In addition to the SRD and SRSW tools which aid if the writing of specifications, the Clarification Notice Tracking System (CTS), provides a communications path between systems engineers, designers, and developers to resolve ambiguities in SRS, IRS, or SSS documents.

Design Tools. PSE has not mandated a specific design methodology, and no specific design tools have been provided. However, commercially available design tools are currently being evaluated, and may be added in the future. Methodologies being considered include Structured Development for Real-Time Systems, Object Oriented Design (OOD), and the Process Abstraction Method for Embedded Large Applications (PAMELA).

One tool which assists in the detailed design process is the Body Builder. This tool parses specifications to create null program bodies. These bodies can be modified to contain a minimal number of statements, allowing the designer to easily create stubs or prototype a system and validate control flow.

Complex Translators. Although there are no complex translators provided as actual tools, there are Ada libraries that contain a syntax analyzer, some generic hashing functions, and a command language interpreter.

Command Interpreters. Other than the basic command language interpreter listed above, no complex command interpreters have been required as part of PSE.

Other PSE Capabilities. Other PSE tools and services include testing tools, a standards checker, an online project information database, reusable software libraries, a graphics editor, and a spelling checker.

Six test tools are in use: the Global Area Peeker/Poker (GAPP), the Input Generator/Output Recorder (IGOR), the Test Data File Generator (TDFG), the Unit Exerciser, and two commercially available test tools. GAPP aids in testing and debugging programs that access shared global areas. IGOR is an interprocess debugging tool that allows recording and viewing of messages sent between processes, and also provides input to processes. The TDFG uses Ada packages and prompts users for input to build test data files for unit testing. The Unit Exerciser is an interactive tool that generates a compilable Ada program (a driver) which exercises a user-specified procedure. The first commercial test tool provides a controlled, repeatable test environment, and is used primarily for regression testing. The second tool, mentioned above, provides coverage and performance data for a program, including statement coverage on a line by line basis. This tool also provides time breakdowns, allowing developers to locate performance bottlenecks.

The standards checker parses compilable Ada code, and provides a report of the code's conformance to project coding standards and guidelines. All source code must be run through the standards checker.

The online project information is accessed through the Programmer's Handbook. It contains a menu driven interface to database entries. Users may read or add entries to the database. Most of the entries deal with helpful "tricks" and otherwise undocumented information of general interest. This promotes the sharing of knowledge between developers in an informal, convenient manner. It also provides an additional online help facility.

There are also online libraries of reusable software resident within PSE. Many of the libraries provide Ada interfaces to COTS software products, screen management functions, and text processing capabilities, among others. There are also functionally oriented packages such as the syntax analyzer, hashing functions, math libraries, and signal processing packages.

The graphics editor and spelling checker are both commercially available software packages.

INCORPORATION OF DOD-STD-2167

In trying to resolve the conflicts between Ada and DOD-STD-2167, GTE encountered the following problems inherent to the defense software industry:

- The available personnel used to staff the PSE development group were largely inexperienced engineers who had to be trained in Ada and in development methodologies.
- There was (and still is) an industry-wide shortage of experienced, qualified Ada engineers.
- Managers at all levels were comfortable with functional decomposition and the waterfall lifecycle, and resisted changing to newer methodologies.
- Managers wanted an environment that would be compatible with the machine-specific development environment used on projects (VAX/VMS).
- Proposal and program managers were reluctant to bid a project using an "untested and unproven" methodology.

With these constraints in mind, GTE decided to follow the methodology, structures, and lifecycle of DOD-STD-2167 closely in the initial development of PSE, with the intention that the environment would evolve as division Ada experience increased. Although many members of the Ada community advised against using DOD-STD-2167 for Ada development, there were advantages to the approach, including:

- A useful, functional APSE was built by relatively inexperienced personnel in a short time span.
- The training provided through the PSE development effort has created a large pool of people who know the syntax and semantics of Ada. Newer personnel have also gained important knowledge of project methodology and development procedures.
- An Ada mindset is slowly evolving at GTE. In particular, people are beginning to realize that Ada is not just "this week's language," but rather the language they will use for years to come. As this mindset continues to evolve, it will become easier to persuade managers that they and their personnel need to be trained to use Ada optimally.

There are also disadvantages in the approach used:

- Because DOD-STD-2167 promotes no specific design methodology, PSE includes very little automated design support.
- Existing code is not nearly as maintainable as it could be.
- Reusable libraries are not very extensive, nor are they well documented.
- Database and graphics capabilities are supported by VAX-specific, non-portable software.

DOD-STD-2167 Incompatibilities with Ada

The following is a summary of perceived incompatibilities between Ada and DOD-STD-2167.[4]

DOD-STD-2167 provides a lot of "how to" information, i.e., users are not only told what must be done, but also how to do it. This improper "how to" information imposes a standard lifecycle, a static program hierarchy, functionally oriented design methods, informal test requirements, use of a PDL, and constraints on reviews. All of these constraints violate DOD Directive 5000.43, the "Acquisition Streamlining" directive.[5]

DOD-STD-2167 also prevents users from taking full advantage of Ada. In general, advanced design methodologies such as OOD, PAMELA, and rapid prototyping go against the methods proposed by DOD-STD-2167. There is also an underlying assumption that all software will be developed specifically for the project, i.e., there will be no reuse. There is also no indication that the developers of DOD-STD-2167 gave any thought to mapping DOD-STD-2167 structures into Ada language structures.

There are also problems with the Data Item Descriptions (DIDs). The DIDs are not Ada oriented, and fail to address the point that Ada may be used as both a design and implementation language. They also require the documentation of information that is redundant or useless. For example, in the detailed design document, input to and output from design components must be listed. However, this information is already clearly documented by the required PDL. Also, the DIDs continue to espouse the default functional decomposition of DOD-STD-2167, and do not allow other methodologies to be easily used.

PSE Resolutions of Incompatibilities

GTE was fortunate in having a customer who did not demand a literal interpretation of DOD-STD-2167, but rather was supportive of GTE's attempts to tailor DOD-STD-2167 for the proper use of Ada. Thus, GTE retained the lifecycle, static hierarchy, terminology, and DIDs described in DOD-STD-2167, but adapted them for use with Ada. Throughout this adaptation process, the PSE development group attempted to provide enough flexibility so all developers would be supported by the environment. The goal was to support new development methods such as OOD, as well as traditional functional decomposition methods.

These areas caused the greatest controversies and required the majority of compromises. The adaptations are described in the sections below.

Adaptations to the Lifecycle. This area required the least modification as most project personnel were already familiar with the "waterfall" lifecycle. Most of the phases specified in DOD-STD-2167 fit into the project phases used at GTE, and are not affected by the use of Ada. The design phases (preliminary design, detailed design) are the only areas where an Ada design method could affect the activities and documentation required by DOD-STD-2167.

Some people in the Ada community assert that when using methods such as rapid prototyping or "code a little, test a little," the distinctions between preliminary design and detailed design are arbitrary and have little meaning. Nonetheless, using separate design phases and reviews[6] has been an important management tool at GTE, and they have been maintained.

Adaptations to the Static Hierarchy. The problems in mapping DOD-STD-2167 static hierarchy components (top level computer software components (TLCSCs), low level computer software components (LLCSCs), and units) to Ada programming units are well known in the Ada community. The only compliant mapping is to allow one or more Ada programming units to correspond to a DOD-STD-2167 unit.[7] GTE has taken this definition and added other guidelines to aid developers in the design of DOD-STD-2167 units:

1. During preliminary design, the software is decomposed into a number of TLCSCs, each of which is represented as an Ada package specification. Whether the decomposition is achieved using OOD (resulting in object TLCSCs) or using functional decomposition (resulting in functional TLCSCs), each TLCSC is defined using an Ada package. Major data items identified during preliminary design are also described as Ada objects.

This use of Ada as a design language during preliminary design is not required by DOD-STD-2167, but promotes consistency of designs, allows preliminary analysis to be performed by compiling the package specifications, and facilitates eventual translation to Ada code.

2. During detailed design, the software is designed completely to the unit level. Unlike preliminary design, there is no restriction on the type of Ada construct used to represent a TLCSC, LLCSC, or unit. All data items are described as Ada objects.

Care must be exercised that designs do not turn into code. Design descriptions should explain what is to be done, but not implementation details.

3. Estimates for source lines of code (SLOC) are used as a management tool throughout the defense software industry. As a very rough guideline, the following SLOC counts are used in selecting software components: TLCSCs - 3000 to 5000 SLOC, LLCSCs - 500 to 2000 SLOC, units - 50 to 150 SLOC.

Due to the fact that a DOD-STD-2167 unit can be represented by more than one Ada programming unit, these guidelines help designers decide when a unit is becoming too large and should become a LLCSC. It is also important to ensure units are logically cohesive. The SLOC guidelines will be adjusted as needed according to project experience.

4. DOD-STD-2167 states that units are the only components of the static hierarchy that will be implemented in code. Because of the flexibility allowed in using any Ada programming unit to represent a CSC in detailed design, this rule does not hold. Insteaci, the PSE uses a convention of identifying all processing (not all code) as either a unit or as 'unit processing' associated with a CSC.

For example, a CSC implemented as a package may include processing that is done at instantiation, or contain a subprogram too small to be considered a unit on its own. In these cases, the processing is identified in the design description as the 'unit processing' part of the CSC.

This labeling has been found to help design reviewers in understanding the structure of the software design.

5. The PSE supports the use of the static hierarchy by incorporating the hierarchical directory structure of the VMS operating system. As the software system is designed, directories are created in the environment to match the components of the static hierarchy.

In addition to the development directory structure, the PSE builds identical directory structures for the use of configuration management and test and integration personnel. Each of these parallel directory structures is called a 'plane.' All development work dealing with a component is done in its corresponding directory in the development plane.

When work on a component for some phase has been completed (the preliminary design, the detailed design, coding, etc.), the source files are 'released' and moved to the corresponding directory in a release plane. Each project defines for itself the number of release planes to be used, who owns them, and how they will be used.

The use of VAX/VMS logical names and search lists (automatically created when the directories are created) make it easy for project personnel to move to any directory (component) in the hierarchy to perform their work.

Adaptations to the DIDs. DOD-STD-2167 describes the design, test, operation, and management documents that must be produced during the life of the software development project. Each of the documents is described in a DID, giving the exact format and content for each section. PSE automates the generation of documents as much as possible, freeing developers to concentrate on the engineering aspects of the software development. The PSE also provides guidance in the type of information to include in the documents.

GTE first followed the tailoring guidelines in DOD-STD-2167 and selected a subset of the DIDs appropriate to the projects. After this tailoring process, the PSE group made selected changes to those DIDs for two reasons: to support the use of Ada as a design and programming language, and to provide a set of design documents that would evolve as the software system evolved.

Using Ada as a design language in both preliminary and detailed design made it impossible to use the original format for the two DOD-STD-2167 design documents: the Software Top Level Design Document (STLDD) and the Software Detailed Design Document (SDDD). In addition, Ada design language is used to describe the data items during the design phases, so the DIDs for the Interface Design Document (IDD) and Data Base Design Document (DBDD) had to be changed.

DOD-STD-2167 calls for the updating of the STLDD and SDDD throughout the life of the project. This involves "backtracking" to make the descriptions of the design components match the final code. Previous experience has shown that this requires much effort and is of little use in the maintenance of the software. Under the PSE, the STLDD and the SDDD are "snapshots" of the software design that are meaningful to the project only at the time they are produced.

After preliminary design, the STLDD is out of date and not maintained, and after detailed design, the SDDD is out of date and not maintained. However, the design information in the STLDD and the SDDD is not lost. It evolves with the software and is maintained in other places.

PSE's adaptation of the design documents breaks them into two major parts: an overview of the software and descriptions of the design components (TLCSCs, LLCSCs, and units). The parts evolve as described below.

The first part, the overview, describes the functionality of the software and how it fits into the rest of the system, the requirements allocated to it, the expected execution characteristics, and the major data used by it. This provides important information for a general understanding of the software as a whole.

After preliminary design, the sections of the STLDD that comprise the software overview are copied, section for section, into the SDDD. During the detailed design phase, they are updated continuously to reflect any changing views of the software. By the end of detailed design, these sections are up to date and contain useful information for the reviewers of the SDDD.

After detailed design, the sections of the SDDD that comprise the software overview are copied, section for section, into the as-built design document, the Software Product Specification (SPS). During the coding, testing and integration phases, these sections are updated continuously to reflect any changing views of the software. By the time the system is delivered, these sections are up to date and contain useful information for maintainers of the software.

The second major part of the design documents, the detailed descriptions of the design components, also evolves with the system. After preliminary design, the TLCSC descriptions from the STLDD are copied into the SDDD, updated with any design changes, and augmented with LLCSC and unit descriptions.

The SDDD may contain textual overview information to describe any large CSCs. This text is in the same format as the overall software overview. After detailed design, any important CSC overview material should be copied into the SPS, in the same manner as the main software overview sections, and maintained in the SPS throughout coding, testing, integration, and maintenance.

At the conclusion of detailed design, the design component descriptions are integrated into the code. Because each of the design components is described using Ada design language, their translation into Ada programming units allows the Ada design language description to be incorporated: the data and interfaces become part of the code, while the structured comments and algorithm description become comments within the code.

In this manner, design information is kept up to date and provides useful information during all stages of the software development, as DOD-STD-2167 requires. In addition, designers, coders, and maintainers are not required to update obsolete documents, supporting the DOD-STD-2167 mandate that the same information not be documented in several different places.

The PSE group wrote their own versions of all the DIDs used, describing the changes to incorporate the use of Ada and the software overview. Like the DOD-STD-2167 DIDs, the PSE DIDs give the exact format and content of each section, and include a complete example of the document.

As mentioned above, the Document Builder toolset supports generation of these documents in the correct format. It builds a skeleton of any of the documents with all the necessary formatting commands and boilerplate text included. The developers need only edit the files to insert the information specific to the project.

For design documents, the tool also supports the generation of the Ada design language descriptions of the components and places those files in the correct directory within the hierarchy. When the document is ready to be generated, the tool collects all the component descriptions from the various directories and composes the document.

TRAINING AND EDUCATION

To ensure that the methodologies and tools provided by the PSE group are used correctly, most training is provided by PSE personnel. Training is performed using five methods: classroom instruction, computer assisted instruction (CAI), videotapes, a user's group, and new project retraining.

Classroom Instruction

Classroom training consists of Ada-related courses, vendor courses, and PSE courses. All classes emphasize hands-on experience with the environment.

Ada Classes. Ada classes are generally taught within projects and are quite often for the benefit of new personnel. These courses give project managers the opportunity to stress those features of the language or design characteristics that are especially important to the project. Conceptual Ada classes, aimed specifically at managers, are also given.

Vendor Classes. Vendor classes are used to train personnel on individual VAX/VMS development tools and other COTS software. Some vendors also offer more generalized classes on design techniques. These are used when appropriate for a particular project.

PSE Classes. The classes taught by the PSE group span a comprehensive range, from management techniques under PSE to software engineering methodologies to tool-specific PSE instruction. Classes range in length from four hours to two weeks, and are given prior to the applicable project phase. The following is a list of some of the courses that PSE has offered or plans to offer:

1. Introduction to Software Engineering Methodology
2. Requirements Analysis
3. Software Design
4. Advanced Software Design

5. Ada Library Management

6. Introduction to PSE

7. Introduction to PSE for Managers

8. PSE Document Generation

9. PSE Requirements Analysis

10. PSE High Level Design

11. PSE Detailed Design

12. PSE Coding and Unit Testing

13. PSE Integration and Testing

14. PSE Configuration Management

15. PSE Software Test Management

These PSE courses cover techniques and usage of tools that provide automated support of the techniques.

Computer Assisted Instruction

GTE uses a commercially available computer-based Ada course as an introduction to the language for both new personnel and persons switching into Ada projects from non-Ada projects. The course is usually completed in approximately eighty hours, and introduces engineers to basic Ada concepts and techniques. The course contains programming exercises on each topic, and is not merely an online textbook. This hands-on approach gives a feeling for Ada and VMS without overwhelming the users.

Videotapes

A set of videotaped Ada lectures is used as an introduction to Ada and software engineering concepts.

User's Group

The PSE group hosts biweekly user's group meetings. This gives the PSE development team a chance to interact with users in an informal environment to exchange ideas and information. The PSE staff provides information on how to use new tools, explain changes in existing tools, and field questions from users.

New Project Retraining

The ability of one group to have such large control over division training and education has an extra added benefit. As older projects wind down and new projects start, people are constantly moving between projects. The use of consistent methodologies and a common tool set can reduce the readjustment period between projects and allow new project personnel to become productive in a shorter time period.

FUTURE DIRECTIONS

The following have been identified as high priority tasks for the PSE group:

1. Fix the problems that currently exist in PSE tools. Some of the tools have major problems in performance, and are not being fully used. In some cases, this may mean a partial or complete redesign and implementation of a tool to increase efficiency. This has been expected, as PSE is still an evolving environment.

2. Update all PSE documentation. The SSPM has recently undergone major updates and when revision A of DOD-STD-2167 is issued, a complete new set of documentation will need to be tailored.

3. Work on filling gaps within PSE. Some areas that could be better supported include automated design support, quality measurement, graphics packages, management and reporting metrics support, PDL support, and hardware/firmware development methodology support. Decisions must continue to be made between developing new tools internally or acquiring COTS software.

4. Consider integrating the tools into a more cohesive environment, possibly defining and using an Ada-oriented command language as the common language for using PSE.

5. Be receptive to the needs of our users. PSE must provide a toolset that meets its users quality and productivity needs; i.e., it must be efficient and effective.

CONCLUSIONS

Although there is still a lot of work to be done on PSE, many lessons have already been learned. A few points that may save others time and effort are listed below:

- There must be system management and hardware support for the development effort. Developers need to have a dedicated system, or at least the authority to use system resources that could impact the entire system.
- An APSE should be developed on a system similar to where it will reside. For example, an APSE to reside on a networked VAX system should not be developed on a single VAX machine.
- When beginning to build an APSE, clearly define a methodology and the APSE requirements as soon as possible. If portions of a methodology cannot be specified, delay defining all but the most generic tools to support that area.
- If the APSE is to support DOD-STD-2167(A), develop it following the standard. This will provide development personnel with experience that other projects can later draw on.
- The APSE development team should use their environment as it becomes available. There is no better test environment than real life usage.
- Train the APSE development team as early as possible, and include training in software engineering principles and the use of DOD-STD-2167.
- If development training must be "on the job," partition the developers into teams and attempt to match personnel so one team member's weakness is another's strength.
- Consider using COTS software whenever an effective and cost efficient tool exists. If development is on a tight schedule, remember that cost efficiency is determined not only by production costs, but by production time as well.

REFERENCES

1. Department of Defense: "Ada Programming Language," American National Standards Institute, ANSI/MIL-STD-1815A, January, 1983.

2. Department of Defense: "Defense System Software Development," DOD-STD-2167, June, 1985.

3. Department of Defense: "Requirements for Ada Programming Support Environments: 'Stoneman,'" February 1980.

4. Firesmith, Donald G.: "Should the DOD Mandate a Standard Software Development Process?," Joint Ada Conference, 1987.

5. Department of Defense Directive 5000.43: "Acquisition Streamlining," January, 1986.

6. Department of Defense: "Technical Reviews and Audits for Systems, Equipments, and Computer Software," MIL-STD-1521B, 1985.

7. Grau, Dr. J. Kaye and Gilroy, Kathleen A.: "Compliant Mappings of Ada Programs to the DOD-STD-2167 Static Structure," Ada Letters, vol. 7, no. 2, March/April 1987.

ACKNOWLEDGMENTS

The author would like to thank all the reviewers of this paper for their time and effort. In particular, he would like to thank Jim Pellmann and Bobbi Poon for the time and ideas they donated to the effort, and Jane Troxel and Lane Parker for their production help.

ABOUT THE AUTHOR

Mr. Lease is a software engineer employed by GTE Government Systems Corporation's Strategic Electronic Defense Division in the Engineering Methodologies/Standards and Computer Support Organization. He has been and is currently involved with development, maintenance, and documentation support for the PSE. Previously, Mr. Lease worked in the areas of Ada programming guidelines development, Ada quality metrics research, and tool development. He may be reached at Mail Stop 5G09, GTE Government Systems Corporation - SEDD, P.O. Box 7188, Mountain View CA, 94039, or by telephone at (415) 966-3439.

Extensibility in an Ada Programming Support Environment

Stowe Boyd, Mark Marcus, and Kirk Sattley

COMPASS

Abstract

The DAPSE (Distributed Ada Programming Support Environment) project[1] is a research effort developing a technology base for mature Ada programming support environments (APSEs). The project is jointly funded by Ft. Monmouth (U.S. Army), Rome Air Development Center (U.S. Air Force), and the STARS program. Other reports on this work have been published [1,2,3].

This report focusses upon extensibility in APSEs and environments in general, and the direction taken in DAPSE in support of extensibility.

Introduction

Consolidating the results of research on integrated programming environments, the DAPSE project applies these results toward the prototyping of a precursor mature APSE. The project is focussed toward a few, high-payoff areas:

- experimentation with a prototype environment and initial set of tools distributed across a local area network of high-performance, raster-graphics workstations;
- investigation of techniques by which a particular DAPSE may communicate with other environments (DAPSEs or otherwise);
- environment support for development and maintenance of Ada systems, and in particular, distributed Ada systems;
- and development of a uniform DAPSE construction methodology, exploiting high-level specification language technology and allied generators to build tool and environment components automatically.

Use of high-level specification languages and allied generators has yielded components of the prototype DAPSE. An initial DAPSE toolset exists in prototype form, and includes

- a graphical "shell" which channels user interaction;
- a distributed Ada application method and supporting, reusable, Ada components which encapsulate system services necessary for inter-processor communication [4,5];
- a rule-based query language, oriented toward Ada intermediate representation and library information [6];
- a family of graphical editors: an Ada structured editor and an Ada library manager based upon grammar-driven graphical manipulation [7];
- tools used in the development of the editors;
- and an integrated configuration control system (CONCON) [2], which manages sets of objects; its primitives include reconciliation (a means to control multiple updates), as well as creation, deletion, and update.

The DAPSE construction methodology is based upon attribute grammar techniques [8,9] and recursive data structures [10]. The methodology has been used to generate the prototype environment, and certain development tools, and is based upon the use of a common specification language. Other generators — the front end generator, data (object and relationship) management system generator, and the structured editor generator — may extend the semantics of the common specification language into their domains. A report describing the use of a uniform specification language, LDO, is in preparation[11].

Barriers to Extensibility

The mature, extensible APSE is a significant departure from the loosely-coupled toolsets currently in use; such an APSE model has been characterized as coherent, or lifecycle-oriented [12]. There are significant barriers to the development and introduction of technology suited to the production of such environments. In particular, the prevalence of tool-oriented methods and environments, and the costs associated with comprehensive environment frameworks stand as the principal sociological barriers. On the technical side, several fundamental problems block the development of extremely extensible environments:

- the current dichotomy between hierarchical file systems and graph-oriented information models,
- the lack of a wide-spread and sufficiently powerful information specification language,
- and the lack of stable interface standards.

A shared theme of the Common APSE Interface Set (CAIS) definition [13] and the Portable Common Tool Environment [14] is "to provide interfaces sufficient to support the use of APSEs for wide classes of projects throughout their lifecycles..." [13; § 2.1]: this is not a sufficient basis for APSE extensibility. In fact, certain features of the CAIS and PCTE designs make uniform extensibility unlikely: in particular, the fundamental split between the "coarse-

[1] DAPSE is funded by Ft. Monmouth, US Army CECOM, with support from The STARS Program Office, and Rome Air Development Center, US Air Force.

Authors' Address: COMPASS, 550 Edgewater Drive, Wakefield MA 01880.
Telephone: (617)245-9540.

grain" information managed by the environment and the "fine-grain" information managed by tools or toolsets.

Immature environments may provide support for the range of activities involved in the software life-cycle, such as requirements analysis, design, deployment, and maintenance. Very few commercial environments have been developed to support extensibility. However, the need for flexibility, extensibility, and modifiability of software support for large-scale systems (>5 million source lines of code) introduces requirements which cannot be satisfied by an immature environment:

- The long lifetime of large-scale systems (over 20 years) requires a highly extensible environment framework.
- Systems of such magnitude may undergo major architectural restructuring due to upgrades and the incorporation of new technology.
- The size and time frame guarantees the need for support of a shifting collection of developers, testers, designers, and support staff.

To date, most efforts toward extensibility in environments have been based upon the UNIX "filter" concept [15]. As an example, consider the Toolpack/Odin project [16,17] in which tool fragments of small capability are composed into tool assemblages in order to carry out specific tasks. Typed information fragments (virtual files or sets of files) are associated with tool fragments, and control of all fragments is centralized in the environment manager. Users are provided a specification language with which to describe new tools or tool fragments upon their introduction into the environment. This facility is more or less the equivalent of defining of new views, and the specification of how those views are created [18]. This approach suffers from the limitations of a 'firewall" between file system and file contents, however. This is similar to the limitations encountered with other systems, such as the IDE environment [19].

In the Arcturus prototype APSE [20], significant effort was directed toward extensibility in user interaction, primarily in the area of user-defined editing presentation and report formatting. This approach does not provide sufficient support in those areas most critical for the management of large-scale software systems.

Extensibility and the Mature APSE

Mature environments provide an adaptable, modifiable structure which supports flexibility within the constraints of certain invariant principles. The DAPSE project has adopted an approach to environment extensibility in which specifications of an intended environment are used to generate an instance of that environment, or to regenerate components of the environment. This approach is quite similar to independent work in the ALMA project [21,22], and the direction originally planned for the WIS Software Development and Maintenance Environment [23].

Extensibility within a DAPSE takes the following form: layers or components of the environment are specified by a high-level, non-procedural specification language: LDO[2]. Software generators have been developed to accept these specifications and automatically produce environment and tool components. (The status of these tools is discussed in more detail in the next section.)

The structure, or schema, of project-specific information managed within such an environment is extensible in an analogous way. The definition of environment objects, such as Ada source, documents, and executables, is described by means of the same specification language. Complex relationships between objects can be represented. The user's interaction with the environment is subsequently channeled through the project-specific characterization of the information managed by the environment.

The integration mechanism which supports this high-degree of extensibility is a partitioned user view provided by a consistent family of editors, as in Gnome [24]. The term editor must be taken in the most general sense, as an interactive mechanism which provides a user with a view over a well-defined information representation, and means to modify the information content of the domain under examination. An alternative to editor is "view manager"; given the definition above, it is clear that a wide number of interactive systems not commonly considered editors are in fact view managers, such as command interpreters, database query languages, Ada library managers, and job control.

Extensibility is not effortless, and the actual process of extending an environment, while automatable, does have a cost. The specification of an information model can be extended to include new objects, relationships and attributes; consider the Ada program library as an example. To support existing tools — such as commercially available Ada compilers — in their manipulation of the library, the extended view needs to be either

1) realized in the form of partitioned information models, one associated with the existing tools' view and the other supporting the auxiliary information,
2) or exploited by the existing Ada compilers.

The goal of the DAPSE approach to extensibility is the second alternative: mature tools should not be "gasketed" into an environmental information model in a passive manner. In the case of an Ada compiler well-suited to a mature APSE, the compiler would share the schema which defines the structure of a program library, and would access the information content based upon the model described there.

This discussion leads to the final, and critical subject: the limits of extensibility. Environments cannot be totally plastic, and the range of extensibility must be well-defined. There are several constraints on extensibility, including

- standards, such as the Ada language requirements for library management;
- costs associated with information and tool transformation caused by modifications in schemata;
- and configuration management overhead imposed by tracking environment versions.

The last point represents the ultimate result of a "metaprogramming" approach [25] where software systems are treated as data to be manipulated. When the environment is extensible, and can change in fundamental ways over the course of a project, the state of the environment must be considered as an

[2] LDO: Language-Derived Objects.

integral part of the state of the project. Attempting to resurrect a previous system version — prior to performing the traditional "build" of system components — will require the "build" of the environment based on the operative schema, followed by the populating of the information model with the system components and supporting materials.

Extensibility in the DAPSE: Current Status

Prototypes of our approach to environment extensibility have been constructed, and reported [1,2,3,4]. The DAPSE project is currently centered around three major releases of demonstration software. This section reports on the state of the second release.

The key to the DAPSE environment technology is information modeling. DAPSE's extensibility centers around the support for the flow, change, and management of newly specified kinds of information within a programming support environment.

In the current DAPSE prototypes, non-procedural specifications describe the schemata for information that are going to be supported by a particular DAPSE environment (e.g. typeset documents, Ada code, Ada libraries, designs, specifications, reports, etc.). These specifications are attribute-grammar based and carry syntactic and semantic content; they allow a system administrator to specify potential relationships among entities stored in the database. For example, a DAPSE might be configured to allow a user to establish a relationship between an Ada compilation unit and a document. Besides allowing relationships between two entities (i.e., coarse grain relationships), fine grain relationships are also supported: for example, an Ada source statement may be related to the pertinent section of a design document.

The remainder of this section outlines the specific DAPSE release 2.0. mechanisms supporting environment extensibility.

Structured Editor Generators

The DAPSE release 2.0 structured editor generator takes as its input

1) a definition of the form of information to be managed by the editor
2) and a mapping which represents the graphical representation of the information structure.

Multiple editors can be generated for the same entity simply by creating different mappings.

The structured editor generator currently generates about 50 percent of the code needed to implement DAPSE editors. With future releases we plan to reach nearly 100 percent of the code. The generators currently generates code to do syntactic checking but not semantic checking. The generated editors take advantage of the high resolution raster graphics based Sun workstation[3].

One of the best features of the generators is their hierarchical nature, and the consistent editor support for suppression of unwanted detail. When editing an Ada package body containing several subprograms, for example, one might first be shown the subprogram names surrounded by boxes (a form of elision) which then can be expanded to show the full bodies on a user's request.

The goal is to produce, as quickly as possible, capabilities for graphically presenting and intelligent editing arbitrary entities. Our current domain comprises data structure- and grammar-based entities, such as Ada source, Ada libraries, command languages, and program management languages. Our hope is that the generated editors will provide sufficient performance, and that the ability to rapidly modify and extend them will offset any apparent productivity loss due to performance.

Entity-Attribute-Relationship Data Base

The DAPSE database system takes as its schema definition language the same specifications of entities that the structured editor generator does. In the current version, the entire set of specifications are combined into a single, very large specification. That specification is used to model the attributes and type system of entities managed by the UNIX file system, as well as relationships between these entities: the coarse-grain information.

Currently, the DAPSE activity window — a DAPSE-specific shell — allows navigation of the database by following relationships between entities. The activity window provides access to both DAPSE and foreign tools in terms of operations defined upon the type system associated with entities. When a DAPSE session starts, all database meta-information is loaded from a set of UNIX files into transient memory.

DAPSE 2.0. database supports a single user. The requirements for a full, multi-user persistent database have not yet solidified, although a prototype multi-user system has been devised, based upon a distributed Ada application paradigm [4,5]. Both the single- and multi-user versions use hand-built code to access a particular database schema, and as such has extremely limited extensibility.

A variety of prototype systems within DAPSE exploit knowledge about the structure of entities. All DAPSE editors do so, in effect making database queries with respect to the entities' internal structures. A prototype query language which allows the user direct query access based upon the internal structure of the Ada program library and Ada program structure has been developed, and is being evaluated as a model for related efforts.

Our hope is that our research will lead us to a model which can merge our fast short-term fine-grain storage facilities with a commercial, persistent, coarse-grain database.

Configuration Control

The configuration control facility of DAPSE provides an orthogonal means of managing the temporal aspect of entities [2,3]. Logically related entities are managed as sets which share a common history. Modification of any entity within a set represents a new instance of the set as a whole[4]. The series of states making up the set history are stored (at the coarse-grain level), and differences between branches in the history can be reconciled.

[3] Sun is a trademark of.Sun Microsystems Inc.

[4] Note that the physical implementation does not require copying the set as a whole.

All access to fine- or coarse-grain information is arbitrated by the configuration control system. The current version of DAPSE is independent of the database schema and therefore extending the schema has no impact on the configuration control mechanisms.

Future Work on Extensibility

DAPSE prototypes have demonstrated the power and flexibility of a specification language/generator approach toward environment technology. Our future goals are a carrying forward of the principles which motivate DAPSE. The increasing automation of graphical view managers (or ed.:rs) will have a large impact on other aspects of the project; in particular, generation of large components within the DAPSE "shell" and the uniform exploitation of the editor paradigm.

A database-oriented policy-management language (along the lines described in [26]) has been defined. In summary, the language extends the basic descriptive capabilities for information models with pre- and post-conditions on database operations. The conditions are planned as the building blocks for software development policy description and enforcement, either interpreted by tools, or used to structure access control.

Environment extensibility must rest upon a stable base, or better, a *collection of stable interfaces*. DAPSE work has led to the identification of non-existent, but needed, interfaces (such as the Ada Library Interface Set (ALIS)[5]) and the need for standard Ada interfaces to other emerging standards (such as POSIX [27] and XWindows [28]). For extensible environments to move out of the laboratory and into the field, widespread commitment to stable interfaces is critical.

Acknowledgements

The DAPSE project is the result of the work of many individuals. Many thanks from the authors to all members of the team, both at COMPASS and Ft. Monmouth.

References

1. M. Marcus, K. Sattley, S.C. Schaffner, E. Albert, "DAPSE: A Distributed Ada Programming Support Environment," Proc. IEEE 2nd Int. Conf. Ada Applications and Environments, 115-125, 1986.
2. M. Marcus, K. Sattley, C.M. Stefanescu, "Configuration Control in an Ada Programming Support Environment," Proc. Joint Conf. National Ada Technology and Washington Ada Symposium, March 1987.
3. S. Boyd, "Status of the DAPSE Project: A Distributed Ada Programming Support Environment," Software Engineering Notes, April 1987.
4. S.A. Schuman, E.M. Clarke Jr., C.N. Nikolau, "Programming Distributed Applications in Ada: A First Approach," Massachusetts Computer Associates, CA-8108-1201, presented at Int. Conf. Parallel Processing, 1981.
5. S. Boyd, H.D. Rombach, C.M. Stefanescu, "DAPSE: A support environment technology for developing distributed Ada programs," in preparation.
6. C.M. Stefanescu, "Using Prolog in a paradigm for prototyping in an APSE," COMPASS CA-8705-1501, May 15 1987.
7. S.C. Schaffner, M. Borkan, "SEGUE: support for distributed graphical interfaces," 21st Hawaii International Conference on System Sciences,1988, in press.
8. N. Habermann, et al. *A Compendium of Gandalf Documentation.* Carnegie Mellon University, 1981.
9. T. Reps. *Generating Language-Based Environments.* MIT Press, 1983.
10. C.A.R. Hoare, "Recursive Data Structures," Int. J. Comp. Inf. Sci., 4(2), June 1975.
11. M. Borkan, S. Boyd, M. Marcus, K. Sattley, S. Schaffner, "A uniform specification language and Ada environment construction," in preparation.
12. J. McDermid, K. Ripken, *Lifecycle support in the Ada environment.* Cambridge University Press, 1984.
13. *DoD Requirements and Design Criteria for the Common APSE Interface Set.* prepared for the Ada Joint Program Office by the KIT/KITIA, 13 December 1985.
14. *PCTE Ada Conceptual Design, Version 2.0*, prepared for the CEC Esprit Program by Mark V Business Systems and Systems Designers PLC, 4 February 1987.
15. E.L. Ivie, "The Programmer's Workbench — A machine for software development," Comm. of ACM 20(10), October 1977.
16. G. Clemm, "ODIN — An extensible software environment: Report and User's Manual," TR CU-CS-262-84, Department of Computer Science, University of Colorado at Boulder, 1984.
17. W. Cowell, L. Osterweil, "The Toolpack/IST programming environment," Proc. SoftFair Conference, Arlington VA, July 1983.
18. W.E. Riddle, J.C. Wileden, "Environment Extensibility — Draft Final," prepared for OUSDRE, IDA paper P-1828, March 1985.
19. A.I. Wasserman, P.A. Pircher. "A graphical, extensible integrated environment for software development," Proc. ACM

[5] Ada Library Interface Standard Birds-of-a-feather. International Ada Conference, Boston MA, December 1987.

SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Palo Alto CA, December 1986.

20. R. Taylor, T. Standish, "Steps to an advanced Ada programming environment," Proc. Seventh International Conf. on Software Engineering, Orlando FL, March 1984.

21. A. van Lamsweerde, et. al., "The Kernal of a Generic Software Development Environment," Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Palo Alto CA, December 1986.

22. A. van Lamsweerde, "ALMA," IEEE Transactions on Software Engineering, in press.

23. S. Boyd, "The WIS Software Development and Maintenance System," Proc. Sixth AFCEA Europe Symposium, Brussels, Belgium, October 1985.

24. D.B. Garlan and P.L. Miller, "GNOME: An introductory programming environment based upon a family of structure editors," Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh PA, April 1984.

25. R.D. Cameron and M.R, Ito, "Grammar-based definition of metaprogramming systems," ACM Trans. on Programming Languages and Systems 6(1), January 1984.

26. L. Osterweil, "A process-object centered view of software environment architecture," DOE report No. DOE/ER/13283-7, December 1986.

27. *POSIX Version 12*, 1987, IEEE 1003 Working Group.

28. *XWindows Version 11*, 1987, Project Athena, MIT.

About the Authors

Stowe Boyd is an Engineering Manager at COMPASS, directing activities in the areas computer-aided software engineering, programming support environments, and large-scale tool integration, including the DAPSE project. He has been a principal investigator in other environment research and development projects, such as the WIS Software Development and Maintenance Environment. Mr. Boyd is quite active in the Ada community with numerous papers published on environments, design methods, and interface standards. He is Chair of the 1988 Future APSE Workshop, Chair of the SIGAda APSE Builder's Working Group, and Co-chair of the IEEE 1000.5 POSIX/Ada Working Group. Mr. Boyd graduated from the University of Massachusetts, Amherst with a B.S. in 1979, and received an M.S. (Computer Science) from Boston University in 1986.

Mark Marcus has been a Systems Analyst at COMPASS since 1980, where he is currently project leader for the DAPSE project. Prior to this project, he led the National Software Works project for a distributed heterogeneous operating system. He is involved in research and development of environment databases, object-oriented development, and advanced presentation media. Mr. Marcus graduated from Tufts University (Electrical Engineering) in 1978.

Kirk Sattley is a Senior Scientist at COMPASS, where he has worked since 1961, principally involved in research and development of Ada environments, compilers, operating systems, and transferable programs. Prior to DAPSE, Mr. Sattley was involved in the design of the Ada Integrated Environment KAPSE, evaluation of Ada candidates, and the design and implementation of the National Software Works project. Mr. Sattley completed his graduate studies at the university of Chicago in 1953.

A PORTABLE SYMBOLIC DEBUGGER FOR DEBUGGING REAL-TIME ADA APPLICATIONS

Elisabeth Broe Christensen

DDC International A/S

Abstract

The paper addresses the suggested topic "Ada Life Cycle Environment: Debuggers". The paper is based on the experience gained during the DDC-I Symbolic Cross Debugger Project. The following topics are addressed: Do we need dedicated Ada debuggers, or can general debuggers be used. Ada features hard to support: Generics and PRAGMA INLINE - how and to what extent are they supported. User friendliness: How this has affected the debugger design. Portability versus minimal interference with the target system - how this is achieved.

Introduction

A debugger is an indispensable tool during the test phase of the Life Cycle for diagnosing program errors. In particular when developing applications, where program failure may be caused by faulty or unstable hardware, a debugger is an invaluable tool for tracking the cause of errors.

A series of issues considered essential for the usefulness of a debugger for the diagnosis of errors in Ada programs are addressed in the following. The paper is based on the experience gained during the DDC-I Symbolic Cross Debugger Project, and examples from this project will be used.

Do We Need Dedicated Ada Debuggers ?

A continuously ongoing discussion is whether the use of Ada requires debuggers developed especially for Ada, or whether general debuggers developed for other languages could be adapted for use with Ada as well.

The opinion of the author is that dedicated Ada debuggers are needed. Ada has some facilities, which were not part of the older languages like Pascal and FORTRAN, for which general debuggers are normally made. These facilities should be supported by the debugger - as well as by any other tool - in order to allow the user to fully exploit the advantages of Ada. We have found that the support of these features of Ada affects the entire debugger design.

What are then these special Ada features ? The first one to be thought of is **tasking**. The tasking model is an integrated part of Ada; an Ada debugger should therefore be able to handle programs with concurrent tasks. This means, that a strategy should be decided for handling the situation where several tasks are or may become breaked (i.e. suspended on breakpoints); and the control structure of the debugger must be prepared for asynchronous events. Furthermore, the debugger should be able to handle the task states as defined by Ada. The current state of a task like "caller in rendezvous", "delayed" and "waiting for children tasks to complete" and state information like current or expected rendezvous partner should be available, as well as information about the task structure like the names of the parent and the children tasks. Furthermore, the debugger should be able to set breakpoints at task events like task activation, rendezvous, termination, abortion and completion.

Some applications do not use parallelism - what about those ? Well, another important feature of Ada is the ability of well-controlled error handling - the exception mechanism. In Ada, the user may define his or her own error conditions and associate a named exception with a given condition. The debugger should be able to set a breakpoint on the raise of a given exception, in order to detect possible error conditions and investigate how they are handled; furthermore, it should be possible when using the debugger to explicitly raise an exception in the user program in order to test the handling of e.g. hardware failures, which may otherwise be difficult to simulate.

Other features which call for dedicated Ada debuggers are

- overloading: A resolution strategy for setting breakpoints on an overloaded procedure must be chosen
- dynamic constraints: This influences the strategy for checking values at assignment
- the package concept: A strategy for looking at entities which are not actually visible from the current point of execution must be designed, as an erroneous value of a variable defined inside some package may affect the system behaviour. The strategy must ensure, that the entity is meaningful (elaborated), otherwise inspection must be prohibited
- generics: A way of handling instantiations must be designed
- PRAGMA INLINE: A way of debugging programs containing INLINE expanded procedures must be invented.

The two latter features: Generics and PRAGMA INLINE, and the problems they cause, are the subject of the next section.

Generics and PRAGMA INLINE

Most Ada debuggers - in fact, to the knowledge of the author, all except the debugger treated in this paper - do not support generics or PRAGMA INLINE. Why are generics and PRAGMA INLINE difficult from a symbolic debugging point of view ?

Basically, because the same source text statement or declaration correspond to several different entities of code, and these different code entities should be distinguishable from the user's point of view !

When doing symbolic debugging, the most frequent activity is to set breakpoints at source text positions - even the most rudimentary debugger supports that. A source text position is referenced by giving a module name, a line number and, in some cases, an identification of which of several potential breakpoint positions on that line you want to break at.

But how do you reference a source text position in a generic instantiation ? As module name the name of the instantiation is used - but what about the line number ? Our choice has been to use the corresponding line number in the generic declaration. From the user's point of view, we think this is the logical thing to do; but it does require more care when translating a source text position into a physical address.

For INLINE expanded procedures you have a problem with the module name. The meaning of a PRAGMA INLINE is to make the procedure code become expanded into - and thus be part of - the code of the entity containing the call. Therefore, the user should be able to distinguish between different expansions of a given procedure. How do you distinguish between different calls of the same INLINE expanded procedure ? Well, you invent a syntax for names allowing you to unambiguously reference a specific call - our choice is to use the source text position of the call as qualifier for the procedure name. Example:

The command

```
SET BREAK AT p #12 INLINE q #5
```

will result in a breakpoint at line number 5 of procedure q, in that specific call to q which is located at line 12 of p.

But this does not solve all problems. According to the definition of Ada, INLINE expansion is only to be carried out when possible - a program is still correct even if some PRAGMA INLINE's have been ignored. This introduces a new set of cases, which must be checked and a handling strategy chosen. The possible cases are:

1) A debugger command specifying an INLINE call is given for a procedure, which has **not** been mentioned in a PRAGMA INLINE.

2) A debugger command specifying an INLINE call is given for a procedure, which was mentioned in a PRAGMA INLINE - but that particular call has not been expanded.

3) A debugger command specifying an INLINE call is given for a procedure, which was mentioned in a PRAGMA INLINE - and the call has been expanded.

4) A debugger command specifying the name of a procedure mentioned in a PRAGMA INLINE is given, but with no indication of which call is referred to, and all calls have been expanded.

5) A debugger command specifying the name of a procedure mentioned in a PRAGMA INLINE is given, but with no indication of which call is referred to; not all calls have been expanded.

Our strategy is as follows: in cases 1,2 and 4 an error is reported. In case 3 the breakpoint is set in the specified call. In case 5 the breakpoint is set in the "shared" procedure code, i.e. the code executed for non-expanded calls.

This is, in our opinion, the best solution from a user point of view. Though from a designer point of view, it complicates the debugger structure: The looking up of names can no longer be separated from the handling of code position information. Whenever a procedure name is looked up, it has to be investigated, whether the name has been mentioned in a PRAGMA INLINE, and if so, whether inline expansion was actually carried out.

We have chosen to support both PRAGMA INLINE and generics as well as any combination of the two; this has affected our entire mechanism for looking up names and deciding visibility. We have made this choice in order to have a useful and user friendly solution. Other aspects of user friendliness are treated in the next section.

User Friendliness

In the recent years, many attempts to define user friendliness have been made. In a way, most of the subjects treated in this paper are related to user friendliness. I will not try to come up with yet another general definition. Instead, I will take a more pragmatic approach: Look at our design goals and at the features of our debugger, and extract those that were not dictated strictly by the debugging operations, but which were added in order to make the debugger easy to use.

The design goals dictated by user friendliness are:

Goal 1: **Minimal User Learning Effort:** The User should be required to learn as little new as possible in order to use the debugger. The User should be allowed to express things in the way he or she is used to.

Goal 2: **Easy Information Access:** The many different pieces of information supplied by a debugger should be presented in a way, which makes it easy for the user to find the relevant information.

Goal 3: **Efficiency in Use:** Frequent operations should require minimal effort.

Goal 4: **Automation:** Work, that may be done by the computer should not be done by the user.

The debugger features reflecting these goals are:

Goal 1: **Minimal User Learning effort:**

- The debugger command language may be tailored to fit the host system conventions: Keywords may be changed and new keywords may be inserted in the debugger commands.

- Command parameters are expressed using Ada syntax. In particular, expressions are identical to Ada expressions.

- When displaying source text defined entities, the definitions are displayed as written by the user, including comments.

- Output (e.g. object values) is formatted to look like Ada source text.

- Use of commands and parameters is consistent.

Goal 2: **Easy Information Access:**

- A set of windows is defined, each dedicated to a particular kind of information. A window needs only to be present on the screen when the commands handling the window information are used. The windows may be moved, closed and have their size changed by the user.

- Display commands have qualifiers indicating the amount of information to be displayed.

- Output (e.g. object values) is formatted to look like Ada source text.

- Default formatting strategies may be set up by the user.

Goal 3: **Efficiency in Use:**

- Function keys are defined for the most frequently used commands.

- Defaults values are defined for certain command parameters.

- Cursors may be used for pointing, i.e. cursor position is used as parameter value.

- Subprograms of debugger commands may be defined.

- Symbols may be defined.

- A facility for easily traversing (and displaying) linked lists is included.

Goal 4: **Automation:**

- Expressions may be used in commands, e.g. when assigning a value to an object or when setting a breakpoint at a physical address.

- Subprograms of debugger commands may be defined.

- Control flow constructs like "if" and "loop" are incorporated in the debugger command language.

- Files may be used for debugger input and output.

Portability, which is treated in the next section, also contributes to achieving Goal 1: "Minimal User Learning Effort": When the User switches from one development system to another, using the same tools minimizes the learning effort.

Portability Versus Minimal Interference with the Target

Portability of a tool is important, because it allows the maximum number of users on various systems to profit from the tool with the minimum amount of time and effort. As stated above, the learning overhead is minimized by using the same tool on different systems; and reuse, which is one of the key concepts of Ada, is facilitated by portability.

The Ada compiler system, of which the debugger referred to in this paper is a part, has been ported to a number of different development systems with hosts ranging from mainframes to mini computers and targets ranging from bare microprocessors to mainframes. Therefore portability is a key issue for this debugger project.

Easy portability of a debugger is not simple to achieve, as a debugger inherently manipulates machine dependent data.

Increased portability of a debugger is often achieved by relaxing the demands for minimum target system interference that are put on the debugger. For example, some portable debuggers rely on calls to a special debugging module to be inserted in the code after each statement. This is not acceptable, in particular not in real-time applications, as the behaviour of the target program is affected whenever the debugger interferes with the target system. Errors may change, or even disappear, because of such interference. The extreme case is when the needs of the debugger cause the target program to grow to a size which makes it impossible to run on the target system.

The debugger referred to in this paper is designed to require minimal interference with the target: no insertions in the target code are necessary. How is this achieved, and what is meant by still stating, that the debugger is easily portable ?

The requirement, that no insertions in the target code must be relied on, means, that address information has to be handled on the host, and, if the debugger is to be easily portable, preferably by the portable parts. This is achieved by defining a set of data structures, on which the portable parts operate. These data structures are built by a portable tool from information which is generated by the the portable parts of the compiler (the front end), propagated and extended with relative address information by the non-portable parts of the compiler (the back end) and finally merged with the information about absolute addresses obtained from the linker.

Information about entity names and visibility is obtained from the symbol table, which is stored in the program library.

The target system interference required is then basically reduced to read and write operations on target memory and registers, and this is only done, when the target program is suspended becausa a breakpoint has been encountered.

Some of the features of the debugger may, though, require changes in the run-time system: If the facility of forcing a task to be suspended, until explicitly released by the user is wanted, an additional task state value may have to be added. But in order to allow for different user requirements depending on the application, and for different levels of system debugging support, the debugger is configurable: When implementing the debugger on a given host/target system, the implementor may choose to implement or leave out any facility. This is yet another aspect of making the debugger easily portable.

So, to summarize, what is meant by stating, that the debugger is easily portable ?

- the maximum amount of debugger processing is carried out by the portable parts

- the target (and host) interfaces are isolated and well-defined

- the data structures for information to be provided by target dependent parts are defined and prepared to be filled out, and a portable tool for building address information is provided

- the debugger is configurable, allowing for different applications and different levels of system debugging support

Conclusion

Ada offers some facilities, which were not part of older languages. The ability to support these facilities affects the entire design of a debugger for Ada; debuggers designed especially for Ada are therefore needed. Some of these facilities are difficult to support. Two examples, namely generics and PRAGMA INLINE, were presented along with the solutions chosen in the DDC-I debugger project.

User friendliness and portability are other important features of a debugger; a set of goals as well as examples of design decisions contributing to these goals were presented.

Elisabeth Broe Christensen

DDC International A/S
Lundtoftevej 1C
DK-2800 Lyngby (Copenhagen)
Denmark

Tlf: +45 2 87 11 44

Elisabeth Broe Christensen is currently project manager for the development of a Portable Symbolic Ada Cross Debugger for the DDC-I Ada® Compiler System®.

After graduating as a M.Sc.E.E (specialized in digital signal processing) from the Technical University of Denmark in January 1982, she worked for 3 years on an advanced medical diagnostic system at Dantec Elektronik A/S, designing and implementing high speed digital hardware, system software, and User Interface Software. To strengthen her background in computer science she studied at the University of Copenhagen, and achieved a B.of Sc. in Computer Science, May 1986.

In May 1985, she joined DDC International A/S, where she became responsible for the first revalidation of the DDC-I Ada Compiler. She participated in the porting of the compiler to the CDC 180 and designed the integrated Program Library Manager for the DDC-I Ada Compiler System. Further, she carried out a study of the tools needed for Ada program development, resulting in a tool plan for DDC-I, of which the debugger is an important part.

®Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

®DDC-I Ada Compiler System is a trademark of DDC International A/S.

A Graphic Design Assistant for Ada® and Information Systems

George Poonen
Helen Martin Nachiappan
Susan Oliver Landstrom

GTE Government Systems Corporation – Wis Division
1 Federal Street, Billerica MA 01821

Abstract

The graphic design assistant (GODzillA) is a tool that facilitates the design of large information systems. It enables a designer to graphically specify procedural as well as data aspects of system design. Procedural design is specified using a modification of an object-oriented notation, whereas the data aspects are specified using an entity-relationship (ER) [CHEN76] notation. The sytem has been partially implemented on a Symbolics™ machine and is currently in use at GTE's WIS Division in Billerica, Massachusetts. Additional features are being incorporated to make the tool more widely applicable. A notable feature of the system is its suitability for very large designs.

1.0 Introduction

This paper discusses a graphic design tool (GODzillA) for designing large information systems. Today "programming in the large" is well recognized as a critical issue. The engineering of large information systems is an equally important issue and requires tools to support both the procedural as well as the data aspects of the design. Therefore, GODzillA embodies a tool to design both large programs in Ada as well as databases using the ER model. In addition, the tool ensures consistency between these two aspects of the design. The focus of this paper is on the description of the Ada tool. This includes a description of the notation developed, the implementation of the tool and how it is currently used.

The standard object-oriented notation, by itself, is inadequate for the design of large Ada programs. The notation developed for GODzillA supports a hierarchical object-oriented scheme allowing for large programs to be manageable and understandable. The system permits both internal and external specifications. The internal specification is stated using a variety of techniques including a finite state machine approach. The system generates Ada specifications including an Ada prologue and Ada body code from a completed design.

Database tools allow the user to rapidly generate error-free, well-designed schemas rather than doing them tediously by hand. This tool allows a user to generate fourth normal form (4NF) schemas from an extended ER model. The standard ER model is insufficient for describing large systems. GODzillA offers some innovative ideas for the expansion of this model. The system generates SQL [DATE86] schemas and enables the coupling of Ada and SQL programs.

The system has been implemented on a Symbolics LISP machine. The interactive environment and the large base of graphics primitives provided the ideal environment for this rapid prototyping effort. GODzillA is currently in use at GTE on the design of two large information systems (greater than 100 entities).

The following sections describe the Ada notation, the database notation and the development of an application using GODzillA. We show the graphic design and the generated code. The paper concludes with our plans for the future. Actual screen dumps of the system are included.

2.0 Graphic Notation for Ada

In 1986 GTE was involved in the design and implementation of a major local area network (LAN). As part of the requirements, it was necessary to document the entire design. Initially we started with a notation described by Booch in [BOOC86]. While this notation was sufficient to describe small systems, the notation was inadequate to describe large designs. During that period

® Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).
™ Symbolics is a registered trademark of Symbolics, Inc.

our research produced a graphic notation which removed some of the limitations [Poon86]. This notation was used successfully to describe the LAN design. However, a significant drawback to the notation was that it was not automated; consequently several inconsistencies in the design went unnoticed. We also noticed limitations in the new notation . Therefore, we decided both to revise the notation as well as implement a graphic tool to express designs. This section briefly describes the notation as it is currently implemented.

In designing a notation for expressing object-oriented designs, we felt the following capabilities were essential:

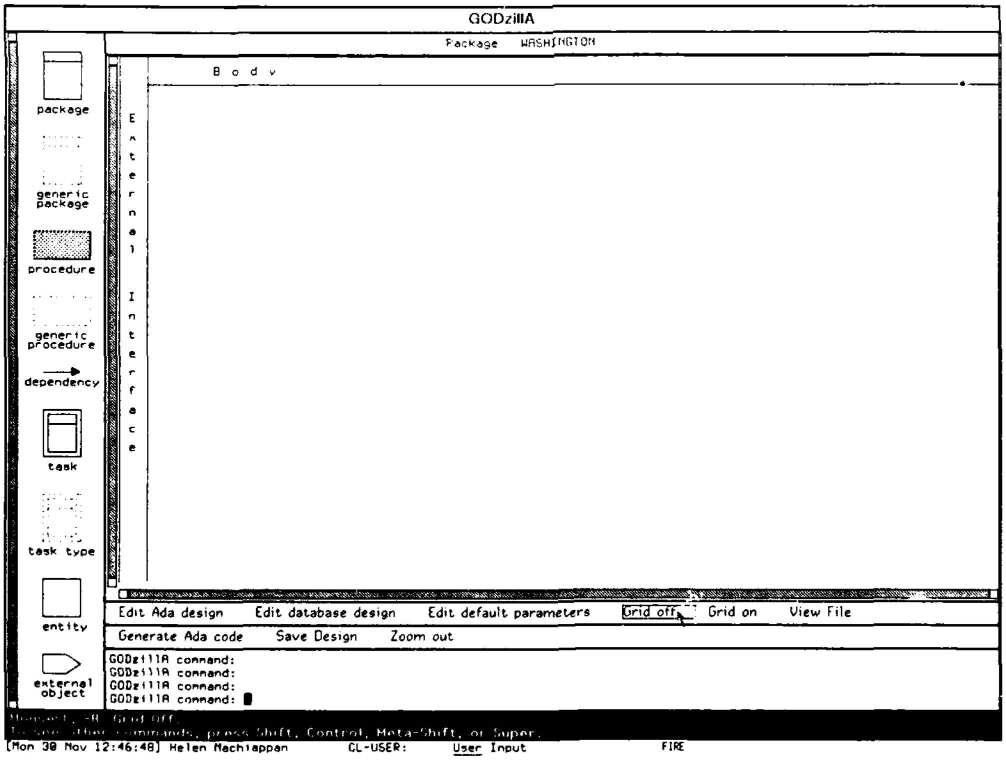
a. the ability to represent large designs.
b. support for decomposition and abstraction.
c. parsimony in expressing designs.

Figure 1 shows the interface the designer sees when beginning a new design. The palette on the left displays the various Ada objects that can be represented. Objects that have no real existence at run time and that are definitional by nature, for example generics or types, are expressed using broken line icons; others are expressed using regular lines. One additional object that is included in the notation is the entity. Entities correspond to database items and are used to connect the procedural design with the database design. The database design is described in Section 4.

Procedure units contain a body and its declared units. Packages and tasks contain an external interface which is connected to the appropriate visible units. A difference between this notation and others is that all units, both internal units and those units externally visible to other programs, are visible to the designer. While internal units are not visible to other programs it is important that the designer have a facility to express to himself the need for these internal units; some of these could be units that are included from a library of useful units. Connections between units indicate dependencies, not data flow.

One of the drawbacks of notations that have been introduced previously [BOOC86, BUHR84] is the clutter they generate for large designs. This detracts from the desirability of a graphic notation. One way to reduce this is to model all objects including the connections as hierarchical objects which could be exploded further. Thus to determine details of the dependencies one merely has to explode the connection. Moreover, associated with each graphic object is a pop-up menu which displays additional attributes of the object.

The body of a procedure or package may also be exploded to show either the Ada PDL asssociated with that unit or a finite state machine (FSM) that may have



```
GODzillA command:
GODzillA command:
GODzillA command:
GODzillA command: █
Mouse-L, -R: Grid Off.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Mon 30 Nov 12:46:48] Helen Nachiappan        CL-USER:      User Input            FIRE
```

Figure 1. Initial State of Ada Designer (GODzillA)

been used to define the behavior of the unit. The FSM is described using a standard notation for transitions and states and is described further in Section 3. GODzillA uses the FSM description to generate the appropriate Ada code.

3.0 Designing an Ada program

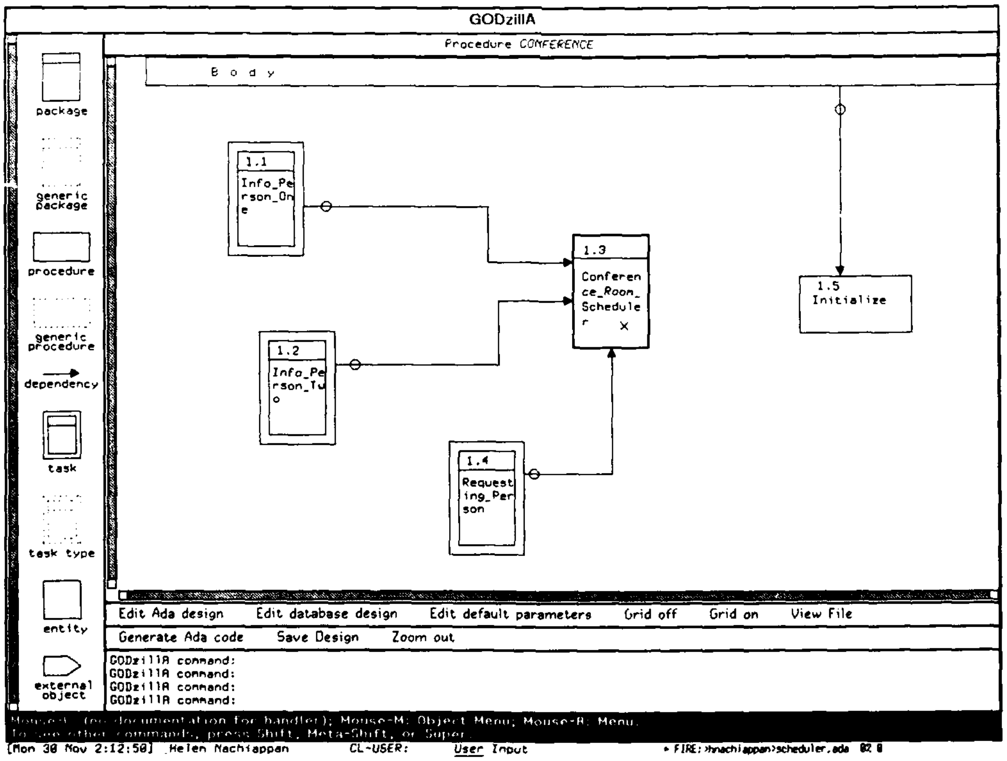
This section describes the use of the Ada notation to describe a simple application. We will first describe the scenario and then show how the designer might create a design using GODzillA.

3.1 Scenario

The problem is to model a conference coordinator who will schedule conference rooms as well as give information on the status of the rooms. The information about the rooms is stored in a database. Several attendees may request information simultaneously about where meetings are being held; others may want to schedule additional meetings.

3.2 Design

For simplicity, we will asume there are only three persons in the system, two of them ask for information about meetings and another requests meeting rooms. Additional people can easily be added to the system without loss of generality. We will model the three persons as Ada tasks INFO_PERSON_ONE, INFO_ PERSON_TWO, REQUESTING_PERSON (for multiple persons we could have used task types), within a main procedure called CONFERENCE as shown in Figure 2. The conference room scheduler manages the conference rooms and is therefore modelled as a package called CONFERENCE_ROOM _SCHEDULER. Notice all Ada units are numbered sequentially. The numbering is used for referencing and traceability purposes. At this stage the designer realizes that all three tasks depend on the CONFERENCE_ROOM_ SCHEDULER and therefore indicates the dependency by connecting the three tasks to the package as shown in Figure 2. Since this is the main procedure, there is no external interface. Moreover, we do not show any external objects for this design. Figure 2 also shows the initialization that needs to be done. The initialization code is defined by a procedure in our case and is called from the body. At this level the designer is not concerned about how the scheduling is done, but primarily about what needs to be modelled. However, since people are asking for information as well updating information, their requests need to be coordinated.



```
GODzillA command:
GODzillA command:
GODzillA command:
GODzillA command:
Mouse-L: (no documentation for handler); Mouse-M: Object Menu; Mouse-R: Menu.
To see other commands, press Shift, Meta-Shift, or Super.
[Mon 30 Nov 2:12:50] Helen Nachiappan        CL-USER:      User Input            • FIRE:>hnachiappan>scheduler.ada 82 0
```

Figure 2. GODzillA representation of Procedure Conference

The first level of design is complete and the designer needs to further elaborate the design of the package CONFERENCE_ROOM _SCHEDULER. He does this by placing the cursor on the package and selecting explode. Visually he gets an empty package with a body and an external interface similar to Figure 1. This package provides two external interfaces, namely GET_INFO and REQUEST_ROOM as shown in Figure 3. Both of these procedures need to access the database entity CONFERENCE_ROOM. Parameters for these access calls can be specified by invoking a menu as shown in Figure 3. For example, procedure REQUEST_ ROOM has five parameters.

As noted above, it is important that simultaneous access to the database be prevented. While several requests to get information are allowed, the conference room entity cannot be updated by more than one task. This requires an internal task, called ROOM_MANAGER to manage the scheduling. Each procedure gets permission from the ROOM_MANAGER to either update or access the CONFERENCE ROOM entity. When the task is completed, the associated procedure reliquishes access so others may gain access. Notice Figure 3 does not depict the details of the scheduler. This is expressed by the designer at the next level of detail as shown in Figure 4.

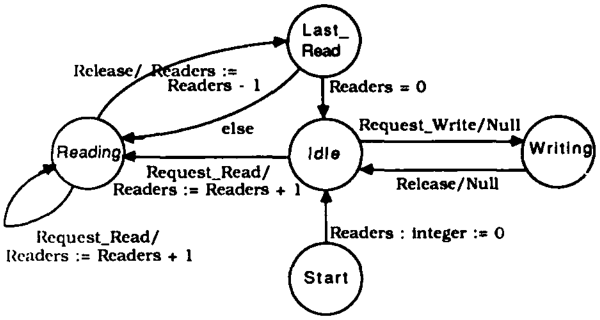
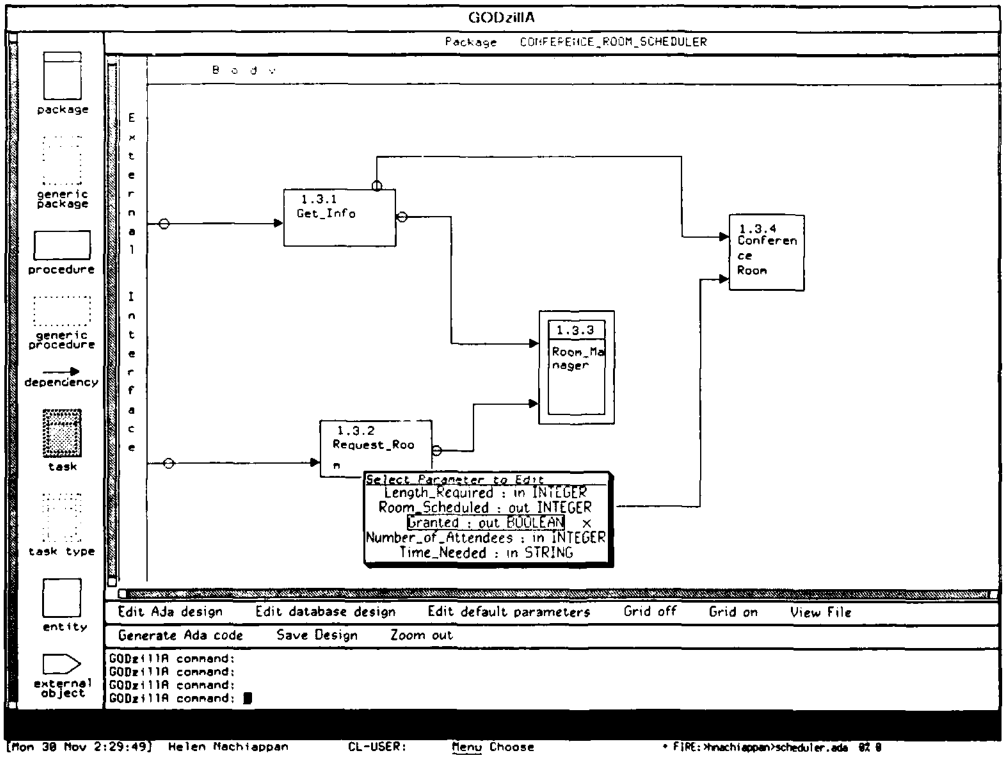

Last_ Read
Release/ Readers := Readers - 1
Readers = 0
else
Request_Write/Null
Reading
Idle
Writing
Request_Read/ Readers := Readers + 1
Release/Null
Readers : integer := 0
Request_Read/ Readers := Readers + 1
Start

Figure 4. FSM for Task Room_Manager

Figure 4 models the scheduling using a finite state machine technique. Circles depict states and arcs depict conditions and actions. Conditions can be either boolean expressions or entry calls initiated by other units. Initially the ROOM_MANAGER task is in the idle state because no requests have been made. Notice in this state a read or a write request can be honored with the appropriate transition being made. If a write request is accepted, the new state is WRITING. For this transition there is no additional action. When the update is completed, a RELEASE is issued by the calling task and the state is again IDLE. With a read request, the number of



```
Select Parameter to Edit
Length_Required : in INTEGER
Room_Scheduled : out INTEGER
Granted : out BOOLEAN   x
Number_of_Attendees : in INTEGER
Time_Needed : in STRING
```

```
GODzillA command:
GODzillA command:
GODzillA command:
GODzillA command: █
```

```
[Mon 30 Nov 2:29:49]  Helen Nachiappan        CL-USER:       Menu Choose                  • FIRE:>hnachiappan>scheduler.ada  0% 0
```

Figure 3. GODzillA Representation of Package Conference_Room_Scheduler

readers is incremented and the new state becomes READING. In this state additional read requests can be honored thus accomodating multiple readers. When the reading is completed, once again a RELEASE is issued by the reading task and the number of readers is decremented. If there are no more readers, the ROOM_MANAGER goes into the IDLE state; otherwise he stays in the reading state. Our notation uses a special state which allows for "If-Then-Else" transitions. To represent initialization, a start state is provided whose arcs only contain actions leading to the first state.

The diagram clearly depicts the asymmetry of the situation with the readers being favored [Buhr84]. The usefulness of the the finite state machine is that requests can be dynamically simulated. With several such machines, potential deadlocks can also be detected.

If the designer is satisfied with the design of the interface and the body, he can generate code directly from the diagrams as shown in Figures 5 and 6. In generating code, we strived to make it readable and as close to what would be written if done manually.The Ada prologue is adapted from DOD Standard 2167 for software documentation.

We have shown a segment of the process that a designer might go through. Other Ada units can be elaborated similarly. The elaboration of the database is done using an extended ER tool. This process is described briefly in the next section. The tool ensures that the entities defined in the Ada design are also captured in the database design and vice versa.

```
package CONFERENCE_ROOM_SCHEDULER is

--ABSTRACT
  --This package schedules reading and writing accesses
    to the conference database.

--KEYWORDS
  --readers, writers

--CONTENTS
  --author: Helen Nachiappan
  --department: CASE
  --created-on: 12/01/87
  --last-revised-on:
  --purpose: To allow the existence of multiple readers
             and writers.
  --function: To prevent simultaneous access to the
              database.

  procedure REQUEST_ROOM(Length_Required : in INTEGER;
                         Room_Scheduled : out INTEGER;
                         Granted : out BOOLEAN;
                         Number_of_Attendees : in INTEGER;
                         Time_Needed : in STRING);

  procedure GET_INFO(Available : out boolean;
                     Time : in Integer;
                     Room : in Integer);

end CONFERENCE_ROOM_SCHEDULER;
```

Figure 5. Generated Specification for Package Conference_Room_Scheduler

```
task body ROOM_MANAGER is
type statetype is (Reading, Idle, Writing);
state : statetype := Idle;
Readers := 0;
  loop
     select
       when state = Idle => accept Request_Write do
         state := Writing;
       end;
     or
       when state /= Writing => accept Request_Read do
         case state is
           when Reading => Readers := Readers + 1;
           when Idle => state := Reading;
                        Readers := Readers + 1;
         end case;
       end;
     or
       when state /= Idle => accept Request_Release do
         case state is
           when Reading => Readers := Readers - 1;
                           if Readers = 0 then
                             state := Idle;
                           end if;
           when Writing => state := Idle;
         end case;
       end;
     or
       terminate;
     end select;
  end loop;
end Room_Manager;
```

Figure 6. Generated Code for Task Room_Manager

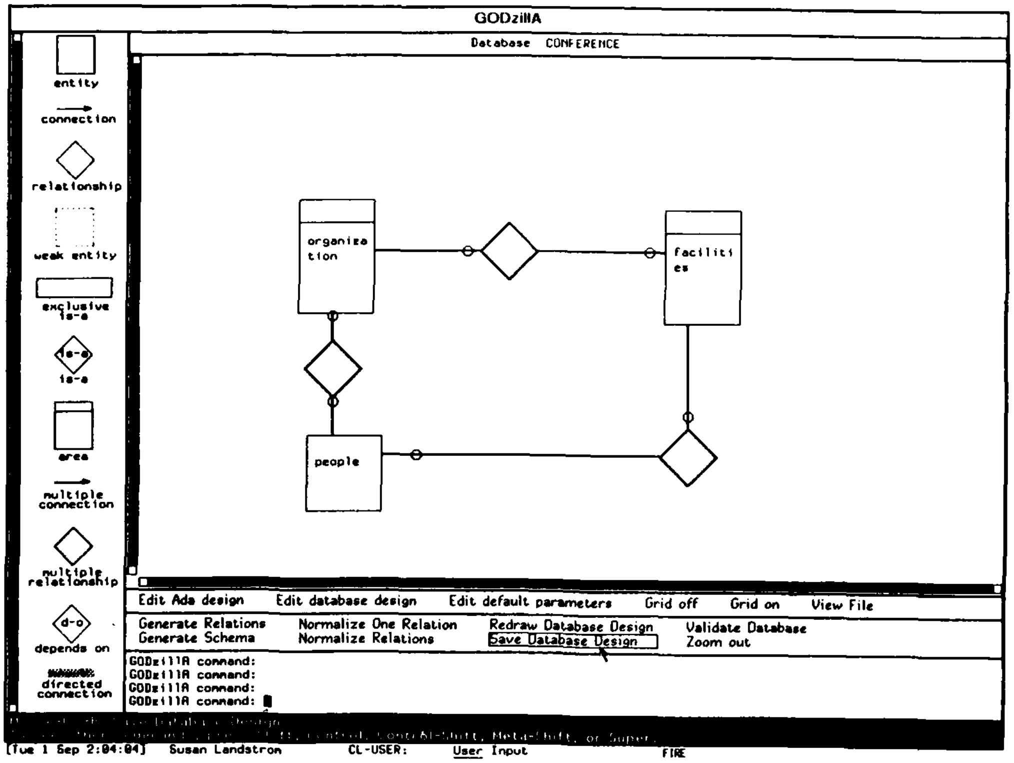
4.0 Database Design

GODzillA includes a database design tool which enables the designer to model databases in terms of an extended ER model. The ER model includes entities shown as rectangles and relationships shown as diamonds. Relationships may be specified as 1 to 1, 1 to many, or many to many. We have extended the model to include several other facilities [MART82] such as:

a. is-a.
b. exclusive is-a.
c. weak entities.
d. areas.

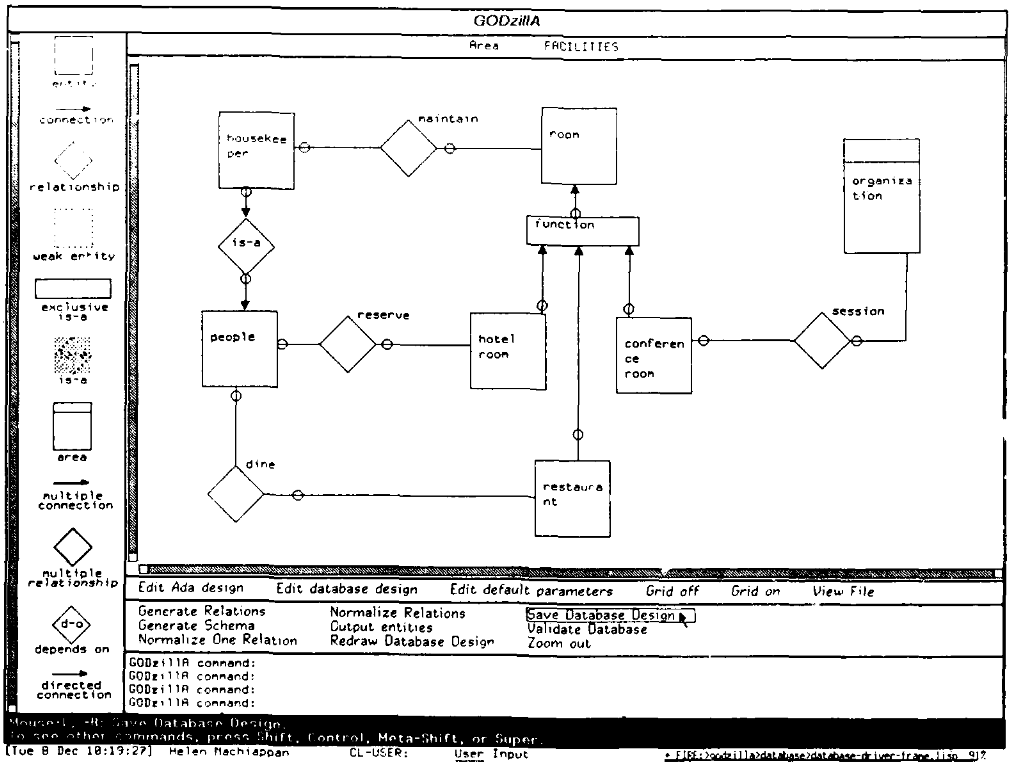
All entities included in the Ada design are automatically included in the database design. The designer may add additional entities and associated relationships. Details of the database design tool are given in [POON87].

Figure 7 shows the top level system design of the database. It includes two areas, ORGANIZATION and FACILITIES, and one entity PEOPLE. Areas are a collection of logically related entities. PEOPLE has several binary relationships with ORGANIZATION and FACILITIES. This is indicated by the bold unlabelled diamond. The entities within ORGANIZATION have several binary relationships with the entities within FACILITIES. To elaborate the areas, the designer explodes that area. Figure 8 shows the explosion of area FACILITIES. The CONFERENCE ROOM entity is part of FACILITIES. Notice HOTEL ROOM, RESTAURANT and CONFERENCE ROOM are depicted



```
GODzillA command:
GODzillA command:
GODzillA command:
GODzillA command:
Mouse-L, -R: Save Database Design.
To see other commands, press Shift, Control, Control-Shift, Meta-Shift, or Super.
[Tue 1 Sep 2:04:04]   Susan Landstrom   CL-USER:   User Input   FIRE
```

Figure 7. A Top-Level System Design Using Areas

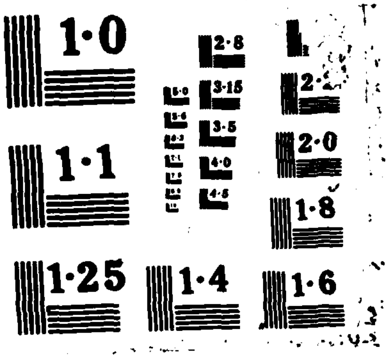


```
GODzillA command:
GODzillA command:
GODzillA command:
GODzillA command:
Mouse-L, -R: Save Database Design.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Tue 8 Dec 10:19:27]   Helen Nachiappan   CL-USER:   User Input   + FIRE:>godzilla>database>database-driver-frame.lisp 91%
```

Figure 8. Decomposition of the Area Facilities

AD-A190 936 PROCEEDINGS OF THE NATIONAL CONFERENCE ON ADA (TRADE 2/7
MARK) TECHNOLOGY (6TH) HELD ON 14-18 MARCH 1988 IN
ARLINGTON VA(U) NORFOLK STATE UNIV VA MAR 88

UNCLASSIFIED F/G 12/5 NL


1·0
2·8
2·5
3·15
3·5
2·0
4·0
4·5
1·1
1·8
1·25
1·4
1·6

as exclusive kinds of rooms, meaning that one room cannot be used as another. Notice that the housekeeper maintains rooms regardless of the function of the room. Thus, we do not have to include three separate relationships to the individual types of rooms. The shaded icons represent entities or areas that are defined at the previous level. Since PEOPLE has a binary relationship with FACILITIES as shown in Figure 7, there must be a binary relationship with PEOPLE and at least one entity in FACILITIES. In Figure 8, PEOPLE is related to three other entities in FACILITIES.

5.0 Conclusion

GODzillA is an operational prototype that was deve:oped as part of an internal research and development project at GTE's WIS Division. The database tools as described above are complete and are being used in two large information system design projects. The Ada tools have implemented the facilities described above, however, the capability to describe all Ada units needs to be included. Currently we have not implemented types and generic facilites although the notation has been defined. We also plan to incorporate the notion of libraries to make the system complete.

Acknowledgements

We are grateful to Tony Christopher for his careful reading of the paper and suggestions for improvements.

References.

BOOC86

Booch, G., "Introduction to Ada," Addison-Wesley, Reading Massachusetts, 1986.

BUHR84

Buhr, R.,"System Design With Ada", Prentice Hall, N.J., 1984.

CHEN76

Chen, P. P.-S., "The Entity-Relationship Model -- Toward a Unified View of Data," *Transactions on Database Systems* **1**, (1), 9-36 (March 1976).

DATE86

Date, C. J., "An Introduction To Database Systems," Addison-Wesley, Reading, Massachusetts, 1986.

MART82

Martin,J., "Design and Strategy for Distributed Processing," Prentice Hall, 1982.

POON86

Poonen, G., Internal Research and Development Report, GTE, WIS Division, August 1986.

POON87

Poonen, G., Landstrom, S., Nachiappan, H., "GODzillA - A Graphic Object-oriented Design Assistant," submitted for presentation at the 10th Annual International Software Engineering Conference to be held in April, 1988.

About the authors:

George Poonen received his B.S and M.S.E.E. degree from the Massachussetts Institute of Technology, Cambridge in 1972. From 1972 to 1974, he was with CCA and SOFTECH . He spent the next eight years with Digital Equipment Corporation where he was responsible for language and database research. He was an independent consultant for various aerospace companies between 1981 and 1986 . Since then he has been with GTE as a senior member of the technical staff.

Helen Martin Nachiappan received a B.A. in German Language Studies and a B.A. in Computer Science from the University of Texas at Austin in 1982 and 1985 respectively. She is currently pursuing her masters in Computer Science at Boston University. She has been employed by GTE since 1986 as a software engineer. During this time she participated in the building of a software development environment for Ada (SDME) before joining the GODzillA project.

Susan Oliver Landstrom received a B.S. in Computer Science from Clemson University in 1983 and an M.S. in Computer Science from Boston University in 1988. Her background includes Computer Aided Engineering, Databases and Artificial Intelligence. She has been a member of the technical staff at GTE since 1986.

ESD Acquisition Support Environment (EASE)

Christopher Byrnes

The MITRE Corporation

ABSTRACT

The ESD Acquisition Support Environment (EASE) is an Ada® analysis and development environment that integrates a variety of off-the-shelf tools into a Sun 3 UNIX workstation running the SunVIEW windowing system. These tools include a Verdix Ada compiler, UNIX textual search utilities and customized Ada unit status monitoring tools. Together these tools aid an analyst in the evaluation of Ada design or code. The analysis that can be performed with EASE today includes checking for strong typing, Ada unit dependencies, initialization and threads of control. This analysis could be done as part of an independent acceptance test of Ada deliverables or as part of internal software quality assurance. The types of analysis which can be done by EASE are being expanded as more Ada analysis tools are developed and integrated into the environment.

1. INTRODUCTION

This document describes the ESD Acquisition Support Environment (EASE) that has been developed at The MITRE Corporation as an aid to the of analysis of Ada deliverables such as Ada as a Program Design Language (PDL) and Ada source code. The growing use of Ada in government software development efforts will lead to a large amount of Ada that should be analyzed as part of a formal design review process. Just as software developers need automated tools to aid in the creation of Ada software, so the analysts of Ada software also need tool support.

EASE was developed as an environment in which to place Ada analysis tools. The tools that are useful to Ada analysis are available from, or under development at, a variety of sources. Some of these tools are Off-The-Shelf (OTS) commercial packages while others are specially developed tools. Because the analysis of Ada covers a wide spectrum, there will be many tools that can be applied to this job.

The current version of EASE provides an integrated environment (running on a UNIX® workstation) into which OTS Ada tools can be placed. As more Ada analysis tools become available or are developed internally within MITRE, they can be added into this environment. EASE has been designed so its suite of tools can be broadened with a minimum of effort.

This document defines the reasons that the analysis of Ada code requires an environment such as EASE and how such an environment should evolve to support future Ada analysis tools. In the interest of space, this document will not describe each feature of the EASE system. The features of EASE are best seen in a demonstration. This document will also outline the future plans for EASE. OTS tools can only perform some types of Ada analysis; MITRE's support work will require much more powerful analysis tools. EASE will evolve to support these new types of analysis tools.

2. OBJECTIVES OF EASE

2.1. FRAMEWORK IN WHICH TO DEVELOP TOOLS

Because we wanted to create an environment with a minimum of development effort that could be extensible, the EASE user interface was developed through the creation of low-level primitives that the higher levels of EASE could be developed on. These primitives provide a layer which abstracts out the details of the workstation environment chosen to implement EASE on. As more OTS tools are added to EASE, these primitives are reused to integrate the new tools.

In addition to abstracting out the workstation dependencies from the tools, these low-level primitives can be used as the basis to implement higher level programming interfaces. The current version of EASE allows programmers to create an environment of OTS tools without having to learn the details of the workstation's conventions. The programmers and maintainers of future versions of EASE will also want to provide sophisticated Ada analysis without first having to learn the details of the current primitive level. This layered approach should make it easier to implement the advanced Ada analysis capability planned for in future versions of EASE.

2.2. CONSISTENT TOOL INTERFACE

EASE provides a consistent interface to all the OTS and developed tools used in Ada analysis. Each tool builder had his/her own conventions for how a user interface should be implemented. In some cases EASE is using tools that were originally part of incompatible environments. EASE supplies the mechanisms to use these tools without having to learn the interface conventions originally provided.

® Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

® UNIX is a trademark of AT&T Bell Laboratories

The need for a consistent interface is particularly acute for Ada analysis groups. An independent Ada analysis group (which could be an internal group such as Quality Assurance or an external group such as MITRE) will usually have only a short time to analyze the delivered Ada. These analysts will not have time to learn or remember the different interfaces of each tool. EASE allows these analysts to concentrate their limited amount of time on actual analysis of Ada deliverables.

2.3. IMPROVED UNDERSTANDING OF SOFTWARE ENGINEERING ENVIRONMENTS

In addition to analyzing Ada software deliverables, MITRE must also review other development documents such as Software Development Plans (SDPs). An important factor in these SDPs is the level of software engineering support provided by a contractor's host environment. The development and use of EASE provides insight into what is needed by a software developer to create and analyze Ada code. This additional insight should be reflected in better analysis of proposed SDPs.

3. EASE ROLE IN THE ACQUISITION LIFECYCLE

3.1. ANALYSIS OF DESIGN PHASE ADA DELIVERABLES

The primary EASE role is to help analyze Ada PDL that is delivered as part of a preliminary or detailed design. This delivered Ada might be part of a Software Requirements Specification (SRS), a Software Top-Level Design Document (STLDD) or a Software Detailed Design Document (SDDD). The SRS, STLDD and SDDD are examples of deliverable documents called for in DOD-STD-2167 which are reviewed in conjunction with a Preliminary Design Review (PDR) or a Critical Design Review (CDR). Recent DoD directives have mandated the use of Ada in both delivered software and as a software design notation. As this mandate is implemented, a greater percentage of delivered designs and source code will be in Ada.

Some of the analysis that can be performed on Ada deliverables is similar to the analysis done on existing natural-language documents. Requirements traceability and allocations of functionality are examples of the traditional analysis that will continue to be done on Ada. But Ada has the constructs to capture a much broader range of design and implementation decisions than normally found in natural-language documents. Analysis of early design (as well as code) deliverables allows the detection of mistakes (such as non-conforming system performance) before expensive coding decisions are made. This additional analysis (provided by the use of Ada) requires additional resources such as those provided by an environment like EASE.

All of this analysis (and all the tools that will be a part of EASE) assumes the use of MIL-STD-1815A Ada, not some subset. All of the analysis tools in EASE must use a common model of semantics and run-time behavior as well as a common syntax. Ada provides such a common model; both developer and analyst can speak the same language. When constrained subsets of Ada are used, important design information may be left out. When alternative models of computation are used, existing tools (such as EASE's) will not understand the design being analyzed.

3.2. ANALYSIS OF ADA PROTOTYPES AND MODELS

In addition to traditional design deliverables such as STLDDs and SDDDs, there is growing interest in requiring other executable products which are designed and implemented in Ada. The use of prototypes, both as a source selection evaluation aid and as an early mock-up of a delivered system, creates software to be checked. Software models can be built to analyze a system's attributes. Ada can be used to write this software which needs the same sorts of analysis that were described above.

3.3. DEVELOPMENT OF ADA CODE

MITRE will also develop its own Ada prototypes and models for its internal use and to support its customers. These programmers will need software environments that support both the development and analysis of Ada code. As a by-product of supporting acquisition, EASE can also be used as an Ada software development environment. Ada developers will be under .he same tight time constraints as Ada analysts and so will need an environment that is just as powerful.

4. ANALYSIS DONE ON ADA DELIVERABLES

4.1. TYPE CHECKING

There is a variety of Ada-specific analyses that can be done on Ada deliverables. One such analysis is of how well typing has been used in the design. Ada is a strongly-typed language; proper use of user-defined data types allows Ada's compilation rules to detect erroneous combinations of data types, operators and objects. An Ada compiler can be used to see if these rules have been followed.

4.2. UNIT COUPLING

The compilation units that make up an Ada program can be analyzed to see the coupling between them. How the units are coupled will depend on the design methodology being used (such as object-oriented or hierarchical) and whether the Ada actually conforms to that methodology. Determining the file compilation (and recompilation) order is a way to determine the coupling between source files. Poorly coupled systems will lead to excessive recompilation and maintenance costs. Such problems should be identified as soon in the development cycle as possible.

4.3. SYSTEM INITIALIZATION

Proper initialization is as important to software systems as steady-state operations. The Ada language has rules defining how Ada units and objects are elaborated (initialized) and the order in which such elaborations are done. Determining the Ada unit elaboration and dependency order allows an analyst to see if the software will initialize properly.

4.4. THREADS OF CONTROL

Analysis of software includes looking at the threads of control under steady state and exceptional conditions. In addition to the compilation rules that define how executable Ada statements can be written, Ada defines the semantic run-time actions of its statements. Ada designs and code can be compiled and executed to see how the threads of control work.

4.5. DESIGN ALTERNATIVES

During the (long) acquisition lifecycle, evolving requirements and improved understanding of target system issues may lead to a need to investigate design alternatives. An analyst may wish to modify a design to see how well it responds to anticipated changes. When working with Ada designs and code, the analyst will want to modify the Ada and then use the same analysis techniques described above on the modified Ada to see how well the design handled the modification. Ada that has be written for maintainability will be able to handle these changes with a minimum of effort. Unmaintainable designs will be discovered by the analysts as they learn how difficult it is to modify such a design.

5. PROBLEMS WITH MANUAL ANALYSIS

5.1. WORKING WITH ADA REQUIRES A "MESSY DESK"

The analysis techniques described above are difficult to perform on paper or using a standard glass (24 lines by 80 characters) terminal. Analyzing Ada requires looking at the specification and body of a unit. Understanding how Ada software works requires looking in both subunits and their parents. This is in addition looking at where subprograms are defined and called as well as where data types are created and used. The information a'out an Ada design is spread throughout the source code, so understanding the source requires looking in many places at once.

Working with just the paper source listings results in the analyst constantly flipping back and forth between pages. Using a glass terminal is only slightly better; without the ability to browse through different source files (and positions within a source file) the analyst will be constantly flipping back and forth between source files. A workstation that provides a "messy desk," i.e., one that allows many simultaneous views into all the Ada source being analyzed, is required. The Smalltalk environment[Gold83] is an example of a workstation providing multiple simultaneous views into the source code. A user of such an environment may "browse" through many portions of the source code at once.

5.2. WORKING WITH ADA REQUIRES COORDINATED EFFORTS

Analyzing Ada deliverables will involve many separate activities, some of which will be occurring simultaneously. In addition to the multiple views of Ada described above, the analyst may wish to run one type of analysis on one part of the Ada (perhaps using the compiler to check strong typing) while doing another type of analysis (perhaps running through a thread of control) on another part. Many of these Ada analysis tools place restrictions on what can and cannot be done simultaneously. Uncoordinated use of these tools can result in the analyst violating these rules.

In addition to coordination among tools, there must be coordination among the Ada source. As the analyst builds or modifies Ada, any changes must be done consistently between the specification and the body, parent and child, etc. Ada's strict interface rules require that any changes must result in recompilation; a large Ada system cannot afford to discover improper updates through trial and error.

5.3. ADA WORK MUST BE DONE IN A PARTICULAR ORDER

There are many steps involved in building (and modifying) an executable Ada design or program. Ada units must be edited, compiled, linked and executed in a particular order. As problems inevitably are discovered, the analyst must remember the error messages produced by one tool in order to go back and see what might have caused the error. A correction and another iteration through the tools may be tried.

The analyst must keep track of all the steps that have been performed and what the results from each step were. These important intermediate results often "disappear" off the top of a glass terminal or they are jotted down on a scrap of paper which is misplaced. Not keeping track of this intermediate information results in wasted time as analysis steps are rerun.

6. EXISTING TOOLS PROVIDE SOME SOLUTIONS

6.1. EXISTING ADA DEVELOPMENT TOOLS CAN BE USED

Many of the analysis techniques described earlier can be performed with existing OTS tools. For example, a validated Ada compiler can be used to check the quality of the Ada code or PDL. The compiler requires that Ada's rules for strong typing and visibility be followed; so the compiler can check how well these Ada attributes have been followed. An analyst can use the Ada compiler to automatically check important aspects of a design without manual intervention.

The Ada library manager must keep track of the compilation and link status of all the Ada source files and units in the program library. The analyst can use the library manager to see if the design can be linked together, an indication that the design/code is complete and consistent. The library manager can also keep track of which tools (compilers and linkers) have been run on Ada source so the state of the library can be determined as tools are run.

An Ada source level debugger can be used to step through the threads of control. By providing test input files and/or having the debugger modify objects, the analyst can see what the threads of control are for different circumstances. A good debugger will also be able to monitor Ada tasking structures and the elaboration/instantiation work done by Ada source as part of system initialization.

Other tools useful in the analysis of Ada source include structured editors, textual search utilities and pretty-printers. The UNIX programming environment provides a rich set of tools, some of which apply to analyzing Ada. The VERDIX Ada Development System (VADS®) provides a set of Ada-specific tools that run on top of UNIX, so version 5.41 of VADS[VERD86] is used as the source for EASE's Ada OTS tools (such as a compiler and library manager).

6.2. EXISTING WORKSTATION WINDOWING SYSTEMS CAN BE USED

In order to support the "messy desk" described earlier, EASE needed a multi-window interface. The SunVIEW® (version 3.0) interface running on Sun 3/75 workstations® (using version 3.0 of Sun's BSD UNIX operating system) was chosen. SunVIEW[Sun86a] provides a fairly standard windowing system that allows multiple windows to be dynamically created and destroyed on Sun's bit-mapped display. These windows can be scrolled, resized, closed into/opened from icons; all from a three-button mouse. Different types of windows support terminal emulation, UNIX shells, text editing, graphics and combinations of all these features.

In addition to a standard set of tools that run on SunVIEW, Sun has defined a 'C' language interface that allows applications software to create their own windows and tools[Sun86b]. EASE uses this SunVIEW interface as the mechanism for providing a multi-window Ada analysis environment. About 13,000 lines of 'C' code interfaces to SunVIEW, the VADS tool set and other UNIX utilities such as "grep."

7. EASE INTEGRATES OTS TOOLS INTO SUNVIEW

7.1. CUSTOM WINDOWS MAINTAIN STATUS OF SOURCE AND UNITS

One example of the customized tools developed specifically for EASE is the windows that maintain the status of Ada source file and compilation units in the program library. The information about a file's or unit's status is gathered from a variety of sources. The program library manager keeps track of which units have been compiled, what source file was used to create a unit and what the specification/body and parent/subunit relationships are between units. The Ada compiler and linker keep track of what source files are currently being compiled/linked and which ones contained errors. The source code editor keeps track of which Ada source files are being edited and which ones have changed (requiring recompilation).

Rather than force a user to check with many different tools to see what the status of the program library is, EASE provides several tools that create and manipulate SunVIEW windows while the analyst is working with the Ada. One window maintains the list of Ada source files in a library (which is a UNIX directory for VADS®) and their status. Another window maintains the list of Ada compilation units, their types (package/subprogram/task, generic/instantiation, parent/subunit, etc.), the associated source file and the unit's status. The status of a file or unit might be "Being Edited," "Up To Date," "Contains Errors," etc. These windows are dynamically updated as the status of a file or unit changes as the result of a tool being run.

7.2. OTS TOOLS INTERFACED THROUGH STANDARDIZED INTERFACES

The OTS tools that an analyst can use on Ada source are complex, with many options that define how a user wishes to use the tool. In following the UNIX convention for defining these options, the tool builders have defined many different command line options that a user must supply (which options are needed will depend on the circumstances). Naturally different tools from different vendors will have their own set of command line options. Even within the same vendor's tool set, the many different tools may have so many options that a user will have difficulty remembering them all.

Learning all the different command line options is particularly important to the Ada analyst. The OTS Ada tools are oriented towards the Ada programmer, so the default behavior of these tools tries to address the needs of the person writing the code. The Ada tool builders often provide capabilities useful to the analyst, but these capabilities are usually placed as obscure command line options. The Ada analyst needs to be able to access these options without having to dig through the back of the Ada tool's reference manual.

EASE has developed a set of customized user dialog boxes for each analysis tool supported. These dialog boxes use SunVIEW fields, toggles and sliders to allow the user to fill in the choice of how a tool will be run. The analyst retains the option of running the tool using the EASE default command line parameters; in that case the analyst does not have to spend any extra time filling in the dialog box.

7.3. OTS TOOLS RUN IN SCROLLABLE WINDOWS

Many of the analysis tools used by EASE will produce textual output that contains useful information about the Ada source file or unit being processed. For example, the Ada compiler will report the positions of any errors in the source. A text searching utility such as "grep" will report the positions of the text that matched the searching criteria. This information is useful to both the analyst and to other EASE tools.

The analyst will want to read and study this information in detail. EASE will capture the textual output from a tool in a scrollable SunVIEW text window, so large amounts of output are not lost off the top of the screen. The analyst may wish to save part of this output for use in documentation and letters; SunVIEW allows the text in these windows to be saved into a file and/or cut and pasted into other windows. Because each invocation of an analysis tool will result in a new EASE window being created, the analyst can save and then compare the outputs of a tool on different versions of an Ada source file.

Other EASE tools can use some of this textual information. An example is the "next error" function provided in EASE, which works similarly to the Emacs[Stal86] "next error" function. The user places the cursor in a window with compiler (or "grep") messages in it and gives the "next error" command. The "next error" function will parse the textual messages until an error message is found. An editor is started up and automatically scrolled to the source line that contains the error. Using this mechanism, an analyst can navigate through a large Ada system and through all the errors the analysis tools have found.

7.4. TILING ALGORITHM MAXIMIZES SCREEN USAGE

The analyst using EASE will be busy looking at all the status windows and the source listings of the Ada being analyzed. Having to worry about defining the position and size of all of the EASE windows that will be dynamically created will only slow the analyst down. EASE implements a simple window tiling scheme that defines the initial size and position of EASE windows. EASE tries to place new windows into currently blank areas of the screen. If all areas are in use, EASE will cover-up the oldest

® SunVIEW and Sun-3 are trademarks of Sun Microsystems, Incorporated

® VADS is a trademark of the Verdix Corporation window. The goal is to maximize screen real estate usage while minimizing user interaction.

7.5. HIDDEN DATABASE MAINTAINS WINDOW RELATIONSHIPS

The status of all the source files, units and windows under the control of EASE is maintained in a hidden database that automatically runs whenever EASE is running. This database allows any tool to query the status of any other tool. The database allows EASE to prevent more than one editor from being run on the same source file or to prevent a file from being edited while it is being compiled. The database automatically shuts down when the user leaves EASE, so the user is never aware of its existence.

The database also maintains the relationships between tools and the windows they are running in. This allows an EASE user to find the window where a particular tool (such as the editor working on a particular source file) is running. The user can use the custom windows managed by this database to find windows and bring them to the top of the display surface, even if that window is deeply buried or closed into an icon. This database brings some organization to the EASE "messy desk."

8. EASE STATUS AND PLANS

8.1. CURRENT EASE STATUS

Currently Version 1.2 of EASE has been released for internal MITRE use. Anyone with the necessary hardware (Sun 3 with 8 MB of memory and SunVIEW 3.0) and software (Version 5.41 of VADS) can use EASE. All of the EASE low-level primitives (as documented in the EASE low-level primitives manual) have been implemented. Experienced UNIX programmers can use these low-level primitives to extend EASE or to implement their own windowing scheme. These EASE primitives can be called from any language that can call 'C', so they can be (and have been) called from Ada programs.

EASE has been used within MITRE to develop a prototype and to evaluate contractor deliverables. It should be noted that EASE requires the mastery of several important skills before its use in analysis work becomes effective. An EASE user must know UNIX, SunVIEW, VADS and EASE conventions in order to run all the analysis tools. EASE only simplifies and standardizes all these interfaces, it does not eliminate them.

8.2. FUTURE PLANS

Future plans for EASE include the integration of more OTS Ada analysis tools as they become available. An example of such a tool is the ANNotated Ada (ANNA)[Luck84] toolset that has already been integrated into EASE. As similar tools (such as those for the Ada Task Sequencing Language (TSL)[Helm85]) become available to MITRE, they will be integrated into EASE.

Currently EASE has only a limited ability to browse through large Ada systems. EASE uses program library information to allow the user to move between specifications and bodies as well as parents and subunits. EASE also uses "grep" for simple browsing; but Ada's scope, renaming and overloading rules make simple textual browsers prone to error. Work has already begun on a DIANA Query Language (DQL) that will allow an analyst to intelligently browse through Ada code, looking for those features or constructs of interest. A DQL proposal describes how this work will be done in detail.

In conjunction with the DQL work, EASE will be used to create a useful set of metrics for analyzing Ada PDL and code. DQL includes the mechanisms for computing metric numbers for Ada constructs. Because DQL will provide access to all of the Ada constructs, metrics can be defined and computed that better reflect the quality of Ada code.

The current version of EASE relies on the analyst to form judgements based on all of the information provided by the various tools. As new tools (such as DQL) are added to EASE, the amount of information available for analysis will grow dramatically. The analyst will need even more powerful mechanisms for managing, displaying and analyzing all this data. Future versions of EASE must be careful not to bury the analyst with too much raw data and not enough information.

One way for future versions of EASE to manage all this information is through the use of expert system shells. An expert system such as the Knowledge Engineering Environment (KEE®)[Inte86] or the Automated Reasoning Tool (ART®)[Clay85] could gather its facts for its knowledge base through DQL and then use its reasoning mechanisms to analyze the Ada code or PDL. The DIANA Query Language Proposal discusses how such Lisp-based expert systems could interface with DQL and EASE. Several expert systems have been ported (or are in the process of being ported) to the Sun 3 workstation so these analysis tools will be available to Sun users soon.

The current version of EASE displays all of its information in textual form. The use of graphical notation would provide a more concise representation of what is in a program library or an Ada package. Several computer scientists have defined graphical notations that can be used to represent the higher levels of Ada code. The Buhr and PAMELA[Cher85] notations are examples of this. Future versions of EASE could add support for these notations. Buhr's CAEDE[Buhr85] system is currently available on a Sun 3 workstation and so it could be used as the basis of EASE's graphical Ada notation.

EASE currently uses the SunVIEW windowing system to provide the graphics interface. Sun has already announced support for SunNeWS®, an improved window support system that could be used to implement the current SunVIEW interface or to implement other standard window interfaces such as the new 'X' standard[Gett86]. As part of the maintenance of EASE, it could be ported to run under SunNeWS[Sun86c] and/or 'X' so EASE might be portable to other 'X' based workstations.

EASE currently uses VADS as the source for all Ada-specific tools such as compilers, linkers and source debuggers. As other UNIX-based Ada compiler systems become available, EASE could be modified to support these compilers as well. The current EASE system has tried to abstract out as much of the compiler dependencies as possible. Part of the maintenance ef-

® KEE is a trademark of IntelliCorp

® ART is a trademark of Inference Corporation

® SunNeWS is a trademark of Sun Microsystems fort for EASE could add support for these alternative Ada compilers.

REFERENCES

[Buhr85] Buhr, R.J.A., et. al., "An Overview and Example of Application of CAEDE," *Ada in use, Proceedings of the Ada International Conference*, Volume V, Issue 2, Association of Computing Machinery, 1985.

[Cher85] Cherry, George, "The PAMELA Methodology, A Process-Oriented Software Development Method for Ada," *Proceedings of SIGAda/AdaJUG Conference*, Association of Computing Machinery, 1985.

[Clay85] Clayton, Bruce, *ART Programming Tutorial*, Inference Corporation, 1985.

[Gett86] Gettys, James, "Problems Implementing Window Systems in UNIX," *Usenix Proceedings*, January, 1986.

[Gold83] Goldberg, Adele, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.

[Helm85] Helmbold, David and Luckham, David, "TSL: Task Sequencing Language," *Ada in use, Proceedings of the Ada International Conference*, Volume V, Issue 2, Association of Computing Machinery, 1985.

[Inte86] IntelliCorp, *KEE Software Development System User's Manual*, IntelliCorp, 1986.

[Luck84] Luckhan, David, et. al., *ANNA, A Language For Annotating Ada Programs*, TR-84-261, Computer Systems Laboratory, Stanford University, 1984.

[Stal86] Stallman, Richard, *GNU Emacs Manual*, Emacs Version 17, 1986.

[Sun86a] Sun Microsystems, *Windows and Window Based Tools: Beginner's Guide*. Sun Microsystems, 1986.

[Sun86b] Sun Microsystems, *SunVIEW Programmer's Guide*, Sun Microsystems, 1986.

[Sun86c] Sun Microsystems, *NeWS Preliminary Technical Overview*, Sun Microsystems, 1986.

[VERD86] VERDIX Corporation, *VADS Version 5.41 Reference Manual*, VERDIX Corporation, 1986.

ACKNOWLEDGMENTS

The author wishes to thank Pamela Simpson for her assistance in programming EASE. The author also wishes to thank Steven Litvintchouck, Richard Hilliard, David Emery and Marlene Hazle for their support and comments during the definition and development of EASE.

LIST OF ACRONYMS

ART	Automated Reasoning Tool
BSD	Berkley Standard Distribution
CAEDE	CArleton Embedded system Design Environment
CCPDSR	Command Center Processing Display Segment Replacement
CDR	Critical Design Review
DIANA	Descriptive Intermediate ANnotation for Ada
DQL	DIANA Query Language
EASE	ESD Acquisition Support Environment
ESD	Electronic Systems Division
grep	global regular expression printer
KEE	Knowledge Engineering Environment
OTS	Off-The-Shelf
PAMELA	Process Abstraction Method for Embedded Large Applications
PDL	Program Design Language
PDR	Preliminary Design Review
QA	Quality Assurance
SDDD	Software Detailed Design Document
SDI EV	Strategic Defense Initiative Experimental Version
SDP	Software Development Plan
SRS	Software Requirements Specification
STLDD	Software Top-Level Design Document
SunNeWS	*Sun Network Extensible Windowing System*
SunVIEW	Sun Visual/Integrated Environment for Workstations
VADS	VERDIX Ada Development System

BIOGRAPHY

Christopher Byrnes is a Member of the Technical Staff of the Software Center of the MITRE Corporation. He received his B.A. from Tufts University in 1976, his B.S. from the University of Lowell in 1980 and his M.S.E. from the Wang Institute of Graduate Studies in 1984. His interest include software development methodologies, analysis of design products and the Ada programming language. His mailing address is: The MITRE Corporation, Burlington Road, M/S A156, Bedford, Mass. 01730. He can also be reached at cb@Mitre-Bedford.ARPA and at ...!decvax!mbunix!cb.UUCP

AN ADA* DEPENDENT FAULT TOLERANT DISTRIBUTED SHORTEST PATH PROGRAM

Pen-Nan Lee and Camron Malik
University of Houston

ABSTRACT

A simple, elegant algorithm upon implementation presents innumerable problems. This paper provides insight into the difficulties of implementing a distributed algorithm. This is followed by a fault tolerant implementation of the Distributed Shortest Path Algorithm. The unrestricted communication in a distributed system produces situations conducive to deadlock. This is particularly true if synchronous message passing is used, as processes may wait indefinitely for each other. To ensure freedom from deadlock dynamic message sending based on Ada timed out entry calls is used. The use of an indirection methodology is also proposed as an alternate to ensure freedom from deadlock.

Distributed programs are also, by virtue of their complexity, difficult to verify. Even after extensive testing residual design inadequacies may be present. Thus the concept of Communication Closed Layers is used to design the program. The Consensus-Global Tester is used to implement error detection and assist in error recovery. These together form Fault Tolerant Layers. In the event of an error, a Backward error recovery scheme is used thus, computation can be reinitiated. The provision of fault tolerance has a large overhead in terms of the number of messages required. A modification of the algorithm is proposed to reduce the number of messages, using buffering in conjunction with Ada constructs to achieve this in the implementation.

I. INTRODUCTION

The trend towards distributed processing on computer networks has led to an increase in the number of distributed algorithms and the development of programming languages to exploit the concurrency. But two major issues have not yet been addressed. The first issue concerns the problems associated with the implementation of the algorithms, within the constraints of a language. The second issue concerns the assurance of reliability in such a complex software system, as the results depend on the unpredictable order in which actions from different processes are executed. In this paper we consider the problems and drawbacks of implementing the Distributed Shortest Path Algorithm [CHAN82] within the constraints placed by a language, specifically Ada*. We then design and implement a fully distributed, fault tolerant program.

The Distributed Shortest Path Algorithm is an elegant distributed solution to compute the shortest path from a special vertex v1 to all other vertices of a weighted, directed graph. The unrestricted communication in a distributed program and the unpredictable order of execution of the component processes pose problems. These are compounded by the constraints placed by a language. Thus to achieve freedom from deadlock requires either an indirection methodology or the use of Ada constructs to provide dynamic output. A general method employing indirection for overcoming deadlock is proposed and implemented using Communication Processes (buffers).

Distributed programs are inherently difficult to verify and even after extensive testing, may have residual design errors. Thus techniques for designing correct programs have to be utilized. This particular fault tolerant implementation is based on the concept of Communication Closed Layers [ELRA83], which partitions programs logically / physically to provide what are called Safe Layers. Such a design methodology coupled with the concept of Consensus-Global Testers [LEE88] provides fault tolerance. Hence, error detection and recovery are possible.

A Recovery Block [RAND75] type scheme is used to implement error detection and recovery. The premise of a Recovery Block type scheme is that errors will occur, thus "spare" modules must be provided. Hence, at the conclusion of a particular computation, if an error is detected the "spare" can be used to recompute the values. While the erroneous values are discarded. The errors are detected through the use of a Tester module which assures that the results are either "acceptable" or erroneous.

The objective is to maximize concurrency and provide fault tolerance without incurring overheads in time-space. To begin the discussion a brief outline of the Distributed Shortest Path Algorithm and backgrounds on the concepts, techniques and methodologies will be given in section 2. This will be followed in section 3 by a description of the implementation. Section 4 is devoted to the analysis. Finally, in section 5, some concluding remarks are made. The outlines of some tasks are provided in the appendix and references are made to the figures in the paper.

II. BACKGROUND

In this section we provide the conceptual background of the techniques which form the basis for this paper. These are firstly, the concept of fault tolerance

*Ada is a registered trademark of the U.S. Govt., AJPO.

followed by the Safe Layering design methodology. Then a consideration of the problems and difficulties of implementing a distributed program are provided. Finally, an overview of the distributed algorithm is given.

Concurrent programs may be executed in several different environments, depending basically on the availability of processors and their interconnections. The first method allows processes to share one or more processors and is referred to as, multiprogramming. If each process is executed on a single processor, but all processors share a common memory, it is referred to as multiprocessing. Finally, the execution of processes on dedicated processors connected by a network is called distributed processing. Since no memory is shared cooperation is achieved through message passing or remote procedure calls. Thus a distributed program consists of a collection of processes or tasks executed in a distributed processing environment.

In what follows, the terms task and process are interchangable and refer to self sufficient execution units which communicate via messages.

2.1 Software Fault Tolerance

The need to provide increased reliability in computer system led to the approach of achieving this goal through the use of fault prevention. Reliance is placed on tools and techniques such as verification, documentation, testing, etc. Such techniques assume that all possible causes of unreliability can be removed prior to delivery and reliance will not be placed on a system until all "bugs" have been removed. This approach fails to account for faults which were unanticipated and thus not weeded out during the design and testing of the system. It is reasonable to assume faults may be present in a system and will have to be tolerated. Thus the concept of fault tolerance uses redundancy of design as a means to provide error detection and recovery from residual design inadequacies. This ensures uninterrupted service even in the event of faults. To achieve this objective, fault tolerant systems must detect errors, assess the damage, try to recover and provide continuous service.

Two complementary approaches for providing fault tolerance in software have evolved. These are forward error recovery and backward error recovery. The aim of forward error recovery is to identify the error and based on the available knowledge correct the system state to provide continued service. An example of such an approach is N-Version Programming. In contrast, backward error recovery manipulates the system state so as to achieve a "reversal of time". That is, to a state prior to the erroneous one without regard for the current state. Thus previous states are saved on a stable medium, to be recalled if the need ever arises.

The recovery block scheme [RAND75] is an example of a backward error recovery technique and like all fault tolerant schemes relies on redundancy. It consists of three distinct parts: a recovery point, execution modules and an acceptance test point. The first of these is a point in the execution of a program when the important variables are saved. This occurs prior to entering a recovery block. The second part consists of a primary module, which is executed first upon entering a recovery block. Upon completion the process must pass an acceptance test to ensure the reliability of its results. If the test is passed, then the process proceeds. But if the test is failed the process state is restored to its original version (saved on entering the recovery block). Then an alternate module of the program is executed, in the hopes that the alternate will not have the residual design inadequacies present in the primary.

The alternate blocks / modules may be of differing design, algorithms, languages or a combination thereof. The premise is that residual design inadequacies present in one module will not be present in another. Any number of alternates may be used as long as they provide a measure of fault tolerance within acceptable costs. For example, if four algorithms to solve a particular problem are available and their time complexities are n log n, n^2, n^3 and n^{12}, then the last version even though it provides redundancy, may be too expensive to employ especially in a time constrained application.

The acceptance test is a last moment check to ensure the reasonableness of the output and is by no means a test for absolute correctness. This acceptance test is over and above the usual interface checks provided by the system - which lead to exceptions, etc. Thus if no exception has been raised and the output of the module meets the acceptance criteria it is assumed that no fault occurred.

2.2 Safe Layering

Distributed programs, by virtue of their complexity, are very difficult to verify formally. Even after extensive testing and debugging residual design inadequacies may be present. This coupled with the unrestricted communication between concurrent processes could cause the propagation of erroneous values. Ultimately leading to erroneous results or a crash of the software system. Thus there is a need for methodologies to design reliable programs and for techniques to detect and recover from faults. One such design method, based on the concept of Communication-Closed Layers proposed in [ELRA83], provides a means to design reliable distributed programs in what are termed Safe Layers. This in conjunction with the Consensus-Global Tester [LEE88] provides error detection and recoverability through Fault Tolerant Layers. The provision of fault tolerance based on these techniques does not give up any degree of concurrency, allowing component processes to execute at their own pace.

2.2.1 Safe Layers

Distributed programs can be viewed as having a two dimensional data-flow. That is, sequential within the process and parallel between processes. Thus, in order to design a distributed program we must consider the sequential behaviour within each process and manage synchronization / communication between the processes. The concept of Safe Layering allows such a consideration. The basic idea is to view distributed programs as a sequential composition of concurrent Layers. For example, a concurrent program P consisting of interacting processes p_1 ; p_2 ; ·····; p_n is defined in CSP [HOAR78] syntax as :

$$P :: [\, p_1 \parallel p_2 \parallel \cdots\cdots \parallel p_n \,]$$

Furthermore, each component process can be subdivided into d logical / physical segments. Thus each process (p_i) may be defined as :

$$p_i :: [\, p_i^1; \cdots\cdots ; p_i^d]$$

Thus, in general, process segments can be defined as :

$$p_i^{seg} \quad \{ i = 1..n \quad , \quad seg = 1..d\}$$

and a Layer^k is :

$$[\, p_1^k \parallel p_2^k \parallel \cdots\cdots \parallel p_n^k \,]$$

The Sequential Composition (denoted by ";") of a concurrent program P is

$$[\text{ Layer}^1 ; \cdots\cdots ; \text{Layer}^d \,]$$

This allows a concurrent program to be viewed as a collection of sequential layers. But gives up some concurrency and requires a global synchronization scheme, as commands in a following layer are not available until the previous layer has terminated.

The Distributed Composition (denoted by ":") of a concurrent program P is:

$$[\text{ Layer}^1 : \cdots\cdots : \text{Layer}^d \,]$$

and is exactly equivalent to :

$$[\, p_1^1; \cdots\cdots ; p_1^d \parallel \cdots\cdots \parallel p_n^1; \cdots\cdots ; p_n^d \,]$$

Thus allowing a process to execute at its own pace without any global synchronization and ignoring layer boundaries.

The equivalence of the two compositions can be provided by assuming for all layers, that Layer^k is Communication Closed. That is, in any communication both members must belong to the same layer. Thus if inter-layer communication is disallowed, across layer boundaries, each of the layers is communication closed and such layers are called Safe Layers. These Safe Layers can be used as units of modularity with layer boundaries serving as synchronization points [LEE88], [ELRA83], [GERT86], [MOIT83].

The Distributed Shortest Path Algorithm (DSPA) is implemented in two layers corresponding to the two phases of the algorithm, described in section 2.4.

2.2.2 Consensus-Global Tester

The efficacy of fault tolerance depends to a large extent on the ability to detect errors and consequently have a chance to correct the errors. Thus error detection is an extremely important phase in computation and relies heavily on the ability of the tester to "catch" the errors. In sequential programs the errors are isolated within single programs which are not affected by outside influences. But in a distributed system, where many processes may be running concurrently and interacting, errors outside the module can affect the outcome. Some errors may be localized but, through interactions, have tainted parts of the program which appear to be fine.

A tester for a sequential program is required to ensure that specifications for a particular program are met. In the case of distributed programs, the tester must ensure the correctness of the results for the entire computation. This is made more difficult since the order of execution, of the actions of the interacting processes, are unpredictable. Thus, so are the results.

The Consensus-Global Tester [LEE88] based on the premise that there are interactions amongst processes provides error detection for all the component processes. This is achieved by providing a global specification, which can test the correctness of the results of all the interacting processes. In the event of a global error all tasks are required to rollback.

If a distributed implementation can be partitioned into regions or layers in such a way that error detection and recovery can be localized. Then the concept of Global Testers can be applied to each of the regions to regionalize the error detection and recovery, without having an adverse effect on the other regions. Thus, errors can be detected and recovery initiated only in those particular regions. In the event of an error, rollback and recovery occur within the region. But if no regional errors are detected the results are sent to the Global Tester for consensus-global testing. That is, to ensure that all regions meet the specification as a whole.

In the program to be implemented the concept of a single Consensus-global Tester for each phase of the computation is used. This tester should verify that the global assertions hold in all cases.

2.3 Problems and Difficulties

The concept of distribut:ng processing is a powerful and useful one, but must be utilized with extreme care. Several problems are faced in the effort to implement a distributed program, and these issues have to be resolved to profit from the enormous potential of distributed processing. These issues include the danger of deadlock, unnecessary blockage, overheads of messages and processes, language constraints, reliability, and a fully distributed implementation. When addressing these issues compromises have to be made, which ultimately affect the implementation and its efficiency.

In distributed solutions, the unrestricted inter-process communication produces situations conducive to deadlock. For example, some arbitrary process P_i attempts to communicate with another process P_j; simultaneously P_j may try to send a message to P_i. This circular wait situation is unresolvable as both processes would wait indefinitely for the other to receive its message. There are two basic solutions to this problem, either deadlock avoidance or deadlock detection and arbitration. The latter is much more costlier in terms of the overhead of monitoring and is almost impossible to achieve, in general, for distributed programs. The avoidance of deadlock is relatively easier to achieve through careful structuring and design of the program [LEE87].

A less serious but equally important issue concerns unnecessary blockage / waiting. A process blocked for communication / synchronization must not have to wait too long. This issue gains significance if it is realized that the speeds of execution of processes are arbitrary and therefore unpredictable. Thus a faster executing process may have to wait for a slower partner to effect a synchronization or complete a communication attempt. For example, if a process P_1 *attempts to communicate* with a process P_2 and finds P_2 busy. P_1 should not be required to wait for P_2, instead P_1 may delay a short time and thereafter proceed on its own, subsequently returning to reattempt a rendezvous.

Since processes are executed on systems which could be geographically separated and no sharing of memory occurs, the only means of communications are remote procedure calls or message passing. In the algorithm and the implementation language, message passing is assumed and thus only the latter is considered. It is apparent that communication through messages has a substantial overhead in terms of the delay, the amount of memory required to buffer message and the number of messages propagated.

Aside from the number of messages, under certain circumstances, the number of processes may be quiet high. These processes may be needed for secondary purposes, such as buffering. They should be kept to a minimum, or eliminated altogether if possible. Since the overhead lies not only in the number of processes but also for inter-process communication. The inefficiency inherent in a system using a large number of processes and / or messages is a drain on the system. This ultimately affects performance and throughput of the system is reduced.

Until the recent development of general purpose programming languages, which incorporate multitasking and constructs for concurrency as primitives, most languages did not provide for such concepts. But the provision of such capabilities in the new languages is by no means complete, as they are still not powerful or expressive enough to allow all types of implementations. In the event that a construct is not directly available to the programmer, the flexibility of a language plays an important role in allowing solutions without incurring unacceptable overheads. An example is the timed out entry call in Ada, without which nondeterminism for output messages would not be possible.The inventors of distributed algorithms usually do not consider specific languages to implement their algorithms. Therefore, these algorithms are not always amenable to implementation within the constraints of a language. The tools provided by a language, either directly or indirectly, may be utilized by a programmer in cases where regular constructs are too confining or inadequate.

There are two types of correctness properties which all programs must possess - Safety and Liveness. Safety properties are the static portion of the specifications and are explicitly stated. An example is mutual exclusion. Liveness deals with the dynamic properties and ensures that an event will eventually happen. Deadlock is an example of a breach of liveness. These issues are extremely important in concurrent programs as the results of the execution of several processes depends on the order in which actions from different processes are executed. The complexity of the situation greatly increases the probability that the programmer will make mistakes and that errors will not be detected during testing. Such design errors would ultimately lead to the violation of the correctness properties and either incorrect results or, failure of the software system. Until reliable proofs of correctness which cover implementation details are available for realistic software, reliance has to be placed on design methodologies and software fault tolerance.

It is obvious that a distributed program must be exactly that, distributed. Since the quality, speed and efficiency all stem from the distributed environment which allows various parts of a concurrent program to execute at their own pace. It is possible to implement distributed algorithms using a host or controller process to restrict communication . But this reduces concurrency and has a detremental effect on the speed, efficiency and ultimately the quality of the program. A centralized model using a single controlling process is infeasible, not only for the reasons above, but it is prone to bottlenecks and intolerant to faults. The loss of the central node can cause a crash of the entire system. Such an implementation would also sequentialize a distributed algorithm, making it no better than a sequential program given time slices on a single processor. Thus all distributed programs must allow unrestricted communication without any host or controller process and use the advantages provided by the language, the algorithm and the system.

2.4 Distributed Shortest Path Algorithm

In this section we provide the background and highlights of the Distributed Shortest Path Algorithm. The complete algorithm can be found in [CHAN82].

The algorithm implemented is an elegant, distributed solution to compute the shortest path from a vertex to all other vertices of a weighted, directed graph in the presence of negative cycles. A directed graph G = (V,E) consists of 2 sets. V is a set of vertices and E is a set of edges. If an edge $<v_i,v_j>$ is incident to vertices v_i and v_j, then a path exists from v_i to v_j. The vertex v_i is called the predecessor of v_j and v_j is the successor of v_i. Each edge has associated with it a length l_{ij} corresponding to the distance from v_i to v_j. In the event a length l_{ij} is negative, a cycle of negative length may exist. Consequently, all vertices reachable from the negative cycle will have l_{ij} equal to $-\infty$. An example of such a graph is shown in figure 1.

In this algorithm processes communicate through messages and the presence of message buffers is assumed. The computation is done in two phases. The first phase computes the minimum distance from vertex v_1 to all other vertices. If there is a negative cycle a vertex will have a distance of $-\infty$. The second phase is used to inform the vertices that they are at a distance of $-\infty$. In phase I the path lengths are propagated using a length message and successors reply using an acknowledgement message.Where there is no ambiguity the terms vertex and node will be used interchangably.

Process P_1 at Node v_1 initiates Phase I by using length messages to inform its successors of its distance from them. The successors upon receiving this value add the distances to their respective successors (to the received value) and pass on the new value.

This iterates until all successors receive their respective length messages. Upon the receipt of a length message each process updates its local value for the shortest path received thus far from a predecessor and propagates the message. An acknowledgement sent in response to a length message is used to terminate phase I.

Phase II, again initiated at node v_1, employs two types of messages. Namely the over- and over? messages. An over- message is sent if it is determined that a negative cycle exists, i.e. shortest path distance is $-\infty$. The receipt of an over- message requires a successor to set distance to $-\infty$, unless it already has distance equal to $-\infty$. The over- message is then propagated. The second message type, an over?, is sent if it has not been determined whether distance is $-\infty$. In the event that there are no outstanding acknowledgements the successor propagates the over?. But, if some length messages remain to be acknowledged, an over- is sent.

The algorithm assumes each process has a queue-like input buffer, to which messages from its neighbors are appended. Since Ada does not support such a capability, one implementation buffers outgoing messages at the source of the communication. The other uses variants of Ada constructs to provide nondeterminism on output.

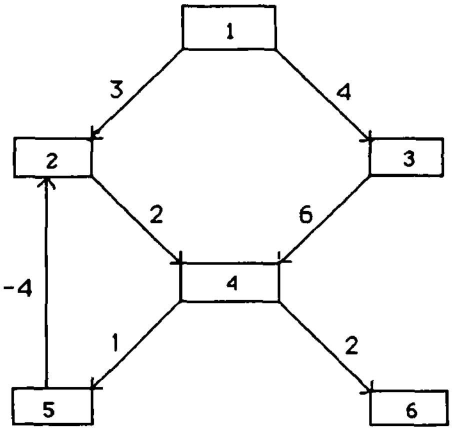


Fig 1. A weighted, directed graph with negative cycle [CHAN82].

III. IMPLEMENTATION

The fault tolerant version of the DSPA program is implemented in two layers corresponding to phases I and II of the computation. The first layer consists of the Primary version and an Alternate for each node of the graph. There is one Tester for each version of each phase which performs error detection and controls the computation, sending the initialization values for each task and receiving the results. The computation is initiated at $node_1$, with each node in the system executing its primary version first. At the conclusion of computation which corresponds to the end of phase I, each of the nodes send its final result (obtained by the execution of the primary version) to the Tester. The Tester verifies that the results are in compliance with the specifications and if no errors are found, the second phase of the computation is started. On the other hand, if the results are found to be erroneous, rollback occurs and recovery is initiated. These correspond to discarding the current values and invoking the alternate version. When the Alternate at each node completes computation, it sends the final values to the Tester for validation. Once again compliance with the specifications is checked and if no errors are detected, the second phase is initiated. Otherwise the computation is aborted, unless more alternate versions are available.

The second phase corresponding to layer 2 consists of two versions, a Primary and an Alternate. Computation is initiated at node 1, with each node executing its primary version for phase II. The sequence of execution and testing is similar to that for phase I. A pictorial representation of the overall structure is shown in figure 2.

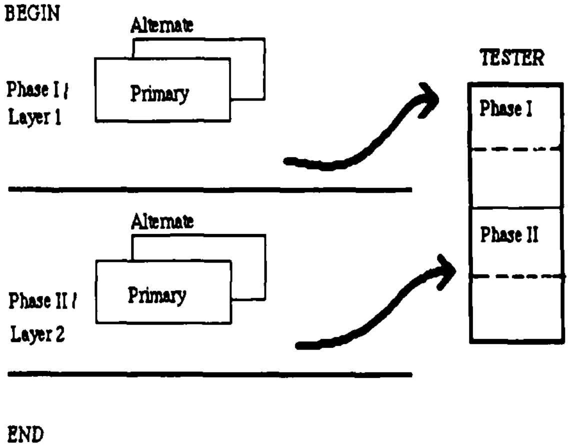


Fig 2. Overall structure of implementation (Layer and Tester)

Each phase / layer consists of a procedure with nested tasks to perform the actual computation. An outline of the overall structure of procedure DSPA is as follows :

```
PROCEDURE DSPA IS

        PROCEDURE Layer1Primary IS
        BEGIN
                  -- layer 1 primary module
        END;

        PROCEDURE Layer2Primary IS
        BEGIN
                  -- layer 2 primary module
        END;

        PROCEDURE Layer1Second IS
        BEGIN
                  -- layer 1 alternate module
        END;

        PROCEDURE Layer2Second IS
        BEGIN
                  -- layer 2 alternate module
        END;

 BEGIN

 END DSPA;
```

The outline of the code for procedure DSPA is shown in figure L of the appendix. An overview of the primary and alternate versions is provided in the following sections.

3.1 Primary Version

The primary version is implemented in two phases similar to the algorithm in [CHAN82]. Each phase consists of a procedure with nested tasks for each node of the graph. These are :

(1) Layer1Primary :: primary version for phase I / layer 1.
- (a) Task L1P1 :: computation task for node1. See appendix figure A.
- (b) Task L1P(i) :: computation task for nodes 2..N See appendix figure B.
- (c) Task Tester :: Tester for phase I. See appendix figure C.

(2) Layer2Primary :: primary version for phase II / layer 2
- (a) Task L2P1 :: computation task for node 1. See appendix figure D.
- (b) Task L2P(i) :: computation tasks for nodes 2..N See appendix figure E.
- (c) Task Tester :: Tester for phase II. See appendix figure F.

A task L1P(i) corresponding to node v_i implements phase I of the algorithm and computes the minimum distance. The shortest path computation is initiated by task L1P1 at node v_1 which sends length messages to its immediate successors and then loops, only accepting messages, until the number of outstanding acknowledgements becomes zero or a length message of less than 0 is received. At which time, it sends a stop message to all its successors. The tasks L1P(i) for all other nodes accept and send messages until they receive the stop message. Each task upon receiving the stop message propagates it until all nodes receive such a message from each of its predecessors. Then all tasks send a copy of their final values for d, pred, num (path, predecessor and outstanding acknowledgements, respectively) to the Tester and complete execution.

The Tester checks compliance with the specifications it is provided. If no errors are found, Phase II is initiated. This involves invoking procedure Layer2Primary, with task L2P1 initiating the computation upon receiving the initialization values from the Tester.

All primary tasks for phase I (L1P(i)) use three types of messages for communicating amongst themselves. The first, a length message, is a triplet (s,Pi,ack) where s is the path length, Pi the source address and ack the acknowledgement for previous length messages. The second is an entry call to entry point STOP, which is used to inform the nodes that phase I has ended. The third is an acknowledgement message (ack) used only to send acknowledgements to the task for node 1 (L1P1).

All tasks upon receiving a length message check whether the path length (s) is shorter than the current shortest path. If so, the tasks compute the values for propagating the message and then buffer them in the Table. The buffering of the shortest path continues until no more tasks are waiting for a rendezvous. At which time the new shortest path is propagated using length messages. If an even shorter path is subsequently received, it is written over the previous shortest path. The use of buffers ensures that only the most minimum of the length messages (of that particular round of messages) will be propagated and requires a buffer size of N - 1 in the worst case. Though a buffer of size N is convenient to declare and use.

During a rendezvous, tasks take the opportunity to return any acknowledgements which may still be owed to the calling task. This is achieved by the use of IN OUT parameters to exchange data. Thus while accepting a length message tasks also return acknowledgements which were buffered along with the previous length messages.

All Phase II tasks, L2P(i), use one type of message with two input parameters consisting of the message type and the task id for communicating among themselves. A message value of 3 signifies an over-, whereas an over? is denoted by a message value of 4. Each task (after receiving the initialization values from the Tester) waits for the initial message from a predecessor at which point it enters a loop which either accepts an over- / over? message, or propagates them. Computation for phase II tasks concludes when over messages from all successors have been received and propagated. At the end of phase II the values for d and over, corresponding to the shortest path and over message, are sent to the Tester for validation. In the event of an error, the Alternate for phase II is invoked under the assumption that phase I is correct. This can be safely assumed because the Tester "passed" the phase I results.

3.2 Alternate Version

The alternate version, invoked in the event of an error by the primary, is implemented as two procedures corresponding to each Phase / Layer. Each procedure consists of three concurrently executing tasks for each vertex v_i of the graph and a Tester. These are:

(1) Layer1Second :: alternate version for Phase I / Layer 1.
(a) Task L1S1 :: alternate for layer 1 node 1. See appendix, figure H.
(b) Task L1S(i) :: alternate for layer 1 nodes 2..N See appendix, figure I.
(c) Task CP(i) :: communication / buffer process. See appendix, figure G.
(d) Task Tester :: Tester for phase I.
(2) Layer2Second :: alternate version for Phase II / Layer 2.
(a) Task L2S1 :: alternate for layer 2 node 1. See appendix, figure J.
(b) Task L2S(i) :: alternates for layer 2 nodes 2..N See appendix, figure K.
(c) Task CP(i) :: communication / buffer process.
(d) Task Tester :: Tester for phase II.

Initialization is controlled by the Tester. The task L1S(i) corresponding to node v_i implements phase I of the algorithm and computes the minimum distance. The tasks L2S(i) implement the second phase and ensure that all over messages are propagated.

Upon receiving the initialization values from the Tester, L1S1 initiates the graph computation sending length messages destined for its successors to its CP_1. The Communication Process (CP_1) in turn redirects them to the destination tasks. If the path received is shorter than the previous one, it is immediately propagated via CP. Otherwise an acknowledgement is sent to the calling task. The computation proceeds similarly to that described for the primary version with the exception that no buffering of messages occurs and all inter-task communication is via the buffer processes (CP). A pictorial representation of the relationship between processes and their CPs is given in figure 3.

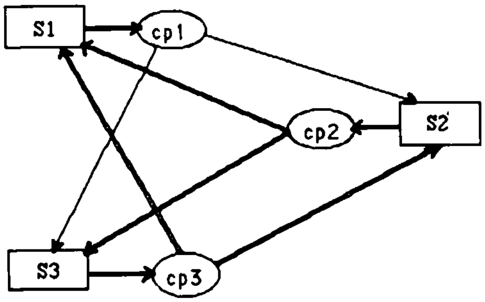


Figure 3. Relationship pathways for processes.

A message from any computation task to its corresponding CP is a 3-tuple (to,mtype,w) which provide the destination address, message type and path length. The CP for phase I tasks can differentiate three types of computation messages depending on the parameter, mtype :

1 :: length message

2 :: acknowledgement message

5 :: stop message.

Whereas CP for phase II tasks can differentiate two types of messages, based on the parameter mtype :

3 :: over- message

4 :: over? message

When a mtype 1 is received the CP redirects it to the destination as a length message 2-tuple (s,Pi) which are the path length and source address, respectively. Upon receiving a mtype 2, an acknowledgement message is sent to the destination. The message types 3 and 4 are used during phase II and correspond to an over- and over?. The final mtype i.e. 5 is sent as a stop message to indicate the termination of Phase I computation. It must be noted that all tasks communicate directly with the Tester, to receive the initialization values and send the final results, thus ensuring reliability.

The layer 2 tasks use one type of message to communicate with each other, that is :

over :: consists of two parameters, mtype and id. A value of 3 for mtype denotes an over- and 4 corresponds to an over?.The id corresponds to the task id.

Each phase II / layer 2 task receives its initialization message from the Tester and is then ready to compute, waiting for task L2S1 (phase II node 1) to initiate the computation. Each task propagates messages until all its successors are notified and then exits the processing loop. Subsequently sending its final values for d and the over message type to the Tester.

In this particular case there are two versions for each phase, but we are not restricted to this. For example, a design similar to the alternate (Second_i) but with buffering of messages at the destination can be used as a third version. Another alternative is the use of different programmers.

3.3 Tester

The Tester for each version is implemented as an Ada task and controls the computation by sending the initialization values to each task. It then receives the results from the computation tasks. When all tasks have responded by sending their final results, the Tester initiates its testing phase which ensures that all specifications are met. If an error is detected the Tester informs the procedure DSPA (using the variable status). Thus, the Alternate for that particular phase can be invoked. If all specifications are met the next phase is initiated or, if it is the last phase, computation successfully completes.

The initialization values for Phase I tasks are:

(1) A boolean list of successors.

(2) A list of lengths to the successors.

(3) The id number of each task (except node 1)

(4) The number of predecessors.

The initialization values for Phase II tasks are:

(1) The shortest path.

(2) The number of outstanding acknowledgements.

(3) A list of successors.

(4) A list of predecessors.

(5) The id number of each task (except node 1)

After ensuring that all the computation tasks have received their initialization values, the Tester waits at an Accept statement for the final values for d, pred, num (path length, predecessor and acknowledgements) from all the phase I computation tasks. It then performs the verification test and sets the variable status accordingly. A status of OK signifies that all tasks passed the test, whereas if status = GE (Global Error) the Alternate will have to be invoked.

A Tester for phase II computations uses a similar strategy to detect errors for phase II tasks. First initializing the tasks and subsequently waiting to receive the values, for the path length and the over message.

The inputs P_I to the Tester at end of phase I :

For all node i :

$$\text{receive } (d_i \wedge \text{num}_i \wedge \text{pred}_i \wedge \text{id}_i)$$

The inputs P_{II} to the Tester at end of phase II :

For all node i :

$$\text{receive } (d_i \wedge \text{over}_i \wedge \text{id}_i)$$

At the end of phase I, a Tester checks whether three assertions are met. These are: if the computation successfully concluded. Secondly, if any negative values for the shortest path are present. If so, either the predecessors shortest path must be negative or the path length must be negative. Finally, whether the shortest path (d) of a node_i is equal to the shortest path of its predecessor plus the length from the predecessor to node_i.

(1) For $\text{node}_1 : \text{num} = 0 \quad \vee \quad s = 0$.

(2) For $\text{node}_i : d_i < 0 \dashrightarrow d_{pred} < 0 \ \vee \ l_{pred,i} < 0$

(3) For $\text{node}_i : \ d_i = d_{pred} + w_{pred,i}$.

At the end of phase II, the Tester checks whether the path length and over message correspond. That is:

(1) For $\text{node}_i : d_i < 0$ --> over- message.

If the conditions are not met then an error is assumed and error recovery is initiated.

IV. ANALYSIS

In the following analysis the difficulties and problems alluded to in section II will be addressed within the context of the construct or methodology used to overcome them. Thus certain issues may be referred to several times, each will provide the technique, construct or methodology used to overcome the problem. The issue of reliability is treated separately.

4.1 Implementation Issues

The bidirectional inter-nodal communication inevitably leads to deadlock in distributed solutions, whereas centralized implementations are too restrictive and intolerant to faults. Reliance was placed on two techniques to overcome the problem of deadlock, these were: the use of an intermediary process (CP) in the alternate version and the use of Ada timed out entry call to provide dynamic output in the primary. The intermediate / buffer process technique avoids deadlock by providing indirection. But poses two major drawbacks, in that, each phase of the implementation requires 2N + 1 (2N tasks + Tester) concurrently executing processes for a graph of N nodes. Secondly, the number of messages also doubles. One message is required from task T_i to the corresponding CP_i and a second from CP_i to the task T_j. Though these drawbacks are associated with the use of intermediary processes, they stem from the constraints placed by the language, which would not allow another deadlock free implementation with such a high degree of parallelism.

Dynamic Sending is the capability for a task to execute an alternate sequence of statements if the called task does not respond to a rendezvous. That is, it is not predetermined that a task will have to wait for its partner in a communication. It may execute alternate statements and at a later time, retry. The timed out entry call allows a sequence of statements to be executed alternatively if an entry call is not accepted within the specified duration. Thus

```
SELECT
   P1.message("entry call");
OR
   DELAY X;
   -- statements
END SELECT;
```

will execute statements following the DELAY, if P1 does not accept the call within X seconds. Consequently, tasks do not need to wait indefinitely for each other. It is worthy to note that the message passing is still synchronous i.e. the called task must respond. There is no message buffering capability. If a rendezvous is unsuccessful it can be attempted later. This allows two-way communication between tasks without resorting to the use of an intermediary process. Thus each phase requires only N + 1 (N tasks + Tester) tasks for a graph of N nodes and a single message suffices for each communication attempt. The problem of deadlocking due to a circular wait situation is no longer an issue. The overhead of such a scheme is the delay incurred in waiting for a task, especially if the rendezvous is unsuccessful. Aside from the benefit of freedom from deadlock, a programmer can specify the time interval to wait for another task.

In Ada, communication is through an entry call made to the called task, which has a corresponding ACCEPT statement. The use of parameters in an entry call allows information to be exchanged by reference or value. The benefit of such a two-way scheme is the ability to exchange length messages and acknowledgements in the same communication. Thus circumventing the need for a task to explicitly send acknowledgements to its predecessors. This was effectively used in the primary version of the implementation, which buffers acknowledgements until the particular task calls with another length message. At which time the acknowledgements are exchanged with the length message. The obvious drawback of this scheme is that acknowledgements are always delayed until the predecessor attempts to communicate. Thus predecessor tasks are always a little "behind" in the information they possess. This is especially true if the task owed the acknowledgements does not communicate again and when computation concludes, the number of outstanding acknowledgements may have an effect on the eventual outcome.

The number of messages propagated in the Alternate implementation is very large. In the worst case N (N - 1 messages + 1 EOT) messages are sent from a task to its CP and the CP propagates N - 1 of those, thus approximately $2N^2$ messages are used for N nodes. The number of messages is large not only for the reason stated above, but also because length messages are propagated even though a following message may provide a shorter path. In the Primary version the use of the COUNT Attribute indirectly provides the capability to reduce the number of messages. The syntax is, P'COUNT, which provides the number of tasks waiting at entry point P and allows the implementation of priorities at a very crude level. Thus tasks can prioritize messages, with in-coming messages having first preference. Outgoing messages are buffered until no tasks are waiting to rendezvous. That is, P'COUNT is equal to 0. This ensures that the shortest path will be propagated after a round of messages and the others will be discarded. In the primary version use of the COUNT Attribute coupled with the modification to buffer messages is instrumental in reducing the number of length messages which are propagated. Considering, that in the worst case, the primary propagates N^2 messages (N nodes each sending N-1 messages) any reduction is a help.

The use of the Communication / Buffer Process (CP) scheme provides an arbitrarily greater degree of concurrency when compared to the primary version. Since tasks, using timed out entry calls, need to delay for a message to get through they are unable to do any thing else. Whereas the CP (Alternate) version sends its messages and can then continue processing. It essentially frees up the task to do something else. In the Primary, the task must itself wait and synchronize with the called task.

The Attribute CALLABLE which returns true if a task is not aborted, terminated or in an abnormal state, was used to aid in message sending. It essentially provided the capability to check a tasks ability to accept messages. Though care must be taken in its use, as a task may infact terminate between the time of the check and the actual message.

4.2 Reliability Issues

Distributed programs as indicated above are difficult to implement and in contrast to sequential programs require the satisfaction of both the safety and liveness properties. Therefore requiring care in the implementation. But this still does not guarantee a correct solution, thus fault tolerant techniques are required to provide some measure of reliability. This reliability can be achieved through careful structuring and design of the program and the use of error detection and recovery techniques. In the implementation of the Distributed Shortest Path Algorithm, the concept of Communication-Closed Layers is used to provide Safe Layers. This was then extended by the use of a Consensus-Global Tester to provide error detection and recovery capabilities.

The use of fault prevention techniques, such as testing, reduce errors but residual design inadequacies may still be present. One method to design programs and provide fault tolerance is the technique of Safe Layering. As described previously, the objective is to partition concurrent programs into concurrently executing segments and to allow communication only within the layers thus created. The Distributed Shortest Path Algorithm by its nature provided an extremely good opportunity to partition it into two layers, corresponding to the two phases of the computation. The logical separation is extended to the physical program, with the provision of two versions for each phase. Error detection is provided through the use of Testers.

The major strength of the DSPA program is its ability to continue processing even in the event of faults. The reliability inherent in fault tolerant software is based on useful redundancy. If the Primary fails each successive alternate version provides continued service but at a degraded level of efficiency or output. The handling of the faults, rollback and recovery, are transparent to the user.

Fault tolerant applications are inherently more inefficient than non-fault tolerant ones. In the case of the DSPA program, the overhead comes from the extra number of messages required to communicate with a Tester. The initialization for each phase requires N messages and N replies are sent to the Tester at the conclusion. If an error is detected the Alternate needs to be initialized, thus N more messages are sent and N received. Consequently, in the worst case 8N messages would be needed and in the best case 4N. This does not take into account the overhead of N messages (minimal) for the stop messages during phase I. But the advantages of fault tolerance are far greater than the drawbacks. For example, fault tolerant software provides continued service, even in the event of faults. In the DSPA program continued service is provided through the use of the Alternate, which will be invoked in the event that the Primary fails to meet its specifications.

An advantage of the Safe Layering technique is that errors caught, lead to a rollback of only that particular layer. Thus valuable time is not lost reinitiating the entire computation. A second advantage is that errors are caught as early as possible in the computation. That is, a fine partitioning allows errors to be detected at the earliest. In the DSPA case, an error detected in phase 2 need only cause a rollback to the begining of layer 2. Secondly, if a fault occurs in phase I it is detected prior to the initiation of phase II. Without Safe Layering the error would be detected at the conclusion of computation, when the test would be performed. It should be noted that the analysis concerning message overheads takes the propagation of a length message into account, not the overall computation.

V. CONCLUSION

It is apparent that distributed algorithms are difficult to implement and are affected by the constraints inherent in the constructs for concurrency provided by Ada. Situations leading to deadlock are pervasive as communication is unrestricted. Whereas, attempts to solve the deadlock problem have tremendous overheads in terms of the number of messages required and the number of processes running concurrently. In addition to the deadlock problem, issues such as memory usage, amount of concurrency and the number of processes executing simultaneously have to be addressed. The solutions to these problems are not easy to find. This puts the burden on the programmer who, as the complexity of the algorithm increases, is more likely to make errors in converting the algorithm to code. His choices will ultimately affect the overall outcome. Incorrect choices may have adverse effects, not only decreasing performance but ultimately leading to problems. Such problems are hard to detect and harder to correct within the confines of the language, especially when trying to maintain a high level of concurrent activity.

In the case of distributed algorithms where the problem of deadlock looms large we can either depend on the programmer and the languages flexibility, or use indirection (as shown by the use of CP in the Alternate implementation). The drawbacks of both are evident. Two methods which would serve better are the extension of the language to provide the necessary features, or deadlock detection followed by arbitration. Since the latter is considerably harder to achieve, a language must provide either the capability to buffer messages implicitly or the ability to send messages nondeterministically. In the event of an inability to implement a deadlock free solution within the constraints of a language, the use of the Indirection methodology is suggested. That is, the use of CP type tasks to ensure freedom from deadlock. This technique will be invaluable in providing a quick and easy solution, while a more elegant one is thought out.

Distributed programs by virtue of their complexity are extremely difficult to verify formally. This is due to the unrestricted communication between interacting processes with unpredictable orders of execution. Thus fault prevention methods are insufficient and reliance must be placed on software fault tolerance, under the assumption that residual design inadequacies are present and may mainfest themselves at some later time.

The use of fault tolerant techniques and the provision of fault tolerance in software provides reliability but at an increased cost, in terms of the messages. But the overhead is minimal compared to the provision of continued service, reliability and the ability to design safe programs, detect errors and correct them.

REFERENCES

ANDE81 Anderson, T. and P.A. Lee, "Fault-Tolerance, Principles and Practice," Prentice-Hall Int., Englewood Cliffs NJ, 1981.

CHAN82 Chandy, K.M. and J. Misra, "Distributed Computations on Graphs: Shortest Path Algorithms," Comm. of ACM,Nov. 1982, vol.25,No.11, pp. 833-837.

ELRA83 Elrad, T. and N. Francez, "Decomposition of Distributed Programs into Communication-Closed Layers," The Science of Computer Programming, No. 2, 1983, pp. 155-173.

ELRA84 Elrad, T.,"A Practical Software Development for Dynamic Testing of Distributed Programs," IEEE Proceedings of the International Conf. on Parallel Processing, Bellaire, MI, Aug. 1984, pp. 388-392.

GERT86 Gerth, R. and L. Shrira,"Proving Noninteraction : An Optimize Approach," submitted to ICACP, 1986.

HOAR78 Hoare, C. A. R.,"Communicating Sequential Processes,"CACM, August 1978, Vol. 21, No. 8, pp.666-677.

LEE87 Lee, P. and C. Malik, "Distributed Shortest Path Algorithm: Constraints and Efficiency Issues in CSP and Ada," To appear ACM South Central Regional Conf, Lafayette, La. Nov 19-2, 1987.

LEE88 Lee, Pen-Nan,"Violation Detection and Recovery of Distributed Programs' Safety Properties," To appear 7th Annual IEEE Phoenix Conf. on Computers and Communications, March 16-18, 1988.

MOIT83 Moitra, A.,"Synthesis of Communicating Processes," Proceedings of the Second Annual ACM Symp. on Principles of Dist. Comp., Montreal, Canada, Aug. 1983, pp. 123-130.

RAND75 Randell, Brian , "System Structure for Software Fault Tolerance," IEEE Trans. on Software Engin., June 1975, Vol. SE-1, No. 2, pp.220-232.

USDD81 U.S. Department of Defense, "Programming Language Ada: Reference Manual," Vol. 106, Lecture Notes in Computer Science, Springer-Verlag, New York, 1981.

APPENDIX

```
ACCEPT Start ( initial message from Tester)
Send length messages to successors
LOOP
  SELECT
    ACCEPT len_msg (length message)
      process length message
      if path received is < 0 then done = true
  OR
    ACCEPT get_ack (acknowledgement message)
      process ack message
      if number of acks = 0 then done = true
  OR
    ACCEPT stop (stop message)
      decrement count of predecessors
  OR
    WHEN done =>
      send stop messages to successors
      if count of predecessors = 0 and all stops sent
          then EXIT
  END SELECT
END LOOP
send final values to Tester
```

Figure A. Outline of Task Primary1 (node1, phase 1)

```
ACCEPT Start (initial message from Tester)
  initialize values
LOOP
  SELECT
    ACCEPT len_msg (length message)
      process length message
      buffer in Table
  OR
      WHEN len_msg'COUNT = 0 =>
        send messages to successors
  OR
      ACCEPT stop message
        decrement count of predecessors; done = true
  OR
      WHEN done =>
        send stop messages to successors
        if count of predecessors = 0 and all stops sent
            EXIT
  END SELECT
END LOOP
send final values to Tester
```

Figure B. Outline of Task Primary (node i=2..N, phase I)

```
initialize values
send initialization values to all phase I (primary) tasks
ACCEPT message0 (message from Task Primary1)
  save reply
ACCEPT message1 (message from Task Primary(i) i=2..N)
If assertion1 then status = OK else status = GE
```

Figure C. Outline of Task GlobalTester (phase I)

```
ACCEPT start (initial message from Tester)
  initialize values
if shortest path < 0 then msg = over- else msg = over?
LOOP
  SELECT
    send over messages if change in message type
  OR
```

```
      WHEN over'COUNT > 0 =>
        ACCEPT over (over- or over? message)
          if message type changes from previous
            change = true
    OR
      if ro change and all over messages received
        EXIT
    END SELECT
  END LOOP
  send final values to Tester
```

Figure D. Outline of Task PrimaryII_1 (node 1, phase II)

```
-- ----------------------------------------------------------------

ACCEPT start (initial message from Tester)
  initialize values
ACCEPT over (initial over message)
  initialize message type
LOOP
  SELECT
    WHEN over'COUNT > 0 =>
      ACCEPT over (over- or over? message)
        if message type changes from previous
          change = true
  OR
    send over messages if change in message type
  OR
    if no change and all over messages received
      EXIT
  END SELECT
END LOOP
send final values to Tester
```

Figure E. Outline of Task PrimaryII (node i=2..N, phase II)

```
-- ----------------------------------------------------------------

initialize values
send initialization values to all phase II tasks
ACCEPT message2 (message from all Tasks PrimaryII)
If assertion2 then status = OK else status = GE
```

Figure F. Outline of Task GlobalTester (phase II)

```
-- ------------------------------------------------------

initialize
ACCEPT idself (id of self from corresponding task i)
LOOP
  SELECT
    ACCEPT msg (from Task i)
      save in Table
    LOOP
      EXIT WHEN end of transmission
      ACCEPT msg (from Task i)
        save in Table
    END LOOP
  OR
    WHEN TRUE =>
      WHILE more messages buffered
         send messages
       compact Table if some messages remain
  OR
    ACCEPT die (terminate message from Task i)
    EXIT
  END SELECT
END LOOP
```

Figure G. Outline of Task CP(i) (node i=1..N , phase I)

```
-- -------------------------------------------------------------

ACCEPT start (initial message from Tester)
send CP its id number
send length messages to CP which redirects to successors
send end of transmission to CP
LOOP
  SELECT
    ACCEPT len_msg (length message)
      process length message
      if path received is < 0 then done = true
    if ack to sent then send to CP
  OR
    ACCEPT get_ack (acknowledgement message)
      process ack message
      if number of acks = 0 then done = true
  OR
    ACCEPT stop (stop message)
      decrement count of predecessors
  OR
    WHEN done =>
      send stop messages to successors
      if count of predecessors = 0
           then EXIT
  END SELECT
END LOOP
send final values to Tester
```

Figure H. Outline of Task Second1 (node1, phase 1)

```
-- ----------------------------------------------------------------

ACCEPT Start (initial message from Tester)
  initialize values
send CP its id number
LOOP
  SELECT
    ACCEPT len_msg (length message)
      process length message
    send length and ack messages to all
    send end of transmission to CP
  OR
    ACCEPT ack_msg (acknowledgement message)
      decrement outstanding acks
    if outstanding acks = 0 send ack to predecessor
  OR
    ACCEPT stop message
      decrement count of predecessors; done = true
  OR
    WHEN done =>
      send stop messages for successors to CP
      send end of transmission to CP
      if count of predecessors = 0
         EXIT
  END SELECT
END LOOP
send final values to Tester
```

Figure I. Outline of Task Second (node i=2..N, phase I)

```
-- ----------------------------------------------------------------

ACCEPT start (initial message from Tester)
  initialize values
send CP its id number
if shortest path < 0 then msg = over- else msg = over?
LOOP
  SELECT
    send over messages if change in message type
  OR
    WHEN over'COUNT > 0 =>
      ACCEPT over (over- or over? message)
        if message type changes from previous
          change = true
  OR
    if no change and all over messages received
      EXIT
  END SELECT
END LOOP
send final values to Tester
```

Figure J. Outline of Task SecondII_1 (node 1, phase II)

-- ---

```
ACCEPT start (initial message from Tester)
  initialize values
send CP its id number
if shortest path < 0 then msg = over- else msg = over?
LOOP
  SELECT
    WHEN over'COUNT > 0 =>
      ACCEPT over (over- or over? message)
        if message type changes from previous
          change = true
  OR
    send over messages if change in message type
    send end of transmission to CP
  OR
    if no change and all over messages received
      EXIT
  END SELECT
END LOOP
send final values to Tester
```

Figure K. Outline of Task SecondII (node i=2..N, phase II)

-- ---

```
status = NR (No Reply)
Layer1Primary
IF status = GE THEN
  status = NR
  Layer1Second
END IF
IF status = OK THEN
  status = NR
  Layer2Primary
  IF status = GE THEN
    status = NR
    Layer2Second
  END IF
END IF
IF status = GE THEN
  GlobalError_and_abort
END IF
```

Figure L. Outline of Procedure DSPA

-- --

AUTOMATED INCORPORATION OF UPSET DETECTION MECHANISMS IN DISTRIBUTED ADA SYSTEMS

Elisa K. Heironimus
Joseph G. Tront

The Bradley Department of Electrical Engineering
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

ABSTRACT

This paper presents an automated approach to developing software that performs single event upset (SEU) detection in distributed Ada systems. SEUs may cause data corruption, leading to a change in program flow or causing a program to execute an infinite loop. Two techniques that detect the presence of these upsets are described. The implementation of these techniques is discussed in relation to the structure of Ada software systems. The block structure of Ada lends itself to the implementation of the detection techniques.

A program, Software Modifier for Upset Detection (SMUD), has been written to automatically modify Ada application software and insert these upset detection mechanisms. The mechanisms have been incorporated into a system model that employs the MIL-STD-1553B communications protocol. This system model is used as a testbed for determining the effectiveness of the detection mechanisms. Ada is used for creating the simulation environment to exercise and verify the protocol. The program, SMUD, is described along with a discussion on the simulation environment and the 1553B protocol. The detection techniques have been tested and verified at the high level using computer simulations. The testing methodology and performance measures are presented.

1.0 INTRODUCTION

This paper presents an automated approach to developing software that performs single event upset (SEU) detection in distributed Ada systems. Upsets considered are those that fall in the transient upset category, i.e., faults that cause no permanent damage to the circuit, but rather cause a perturbation of data or control information. Transient upsets may cause information corruption, leading to a change in program flow or causing a program to execute an infinite loop. Two techniques that detect these undesirable events are described in this paper. The implementation of these techniques is discussed in relation to the structure of Ada software systems.

The focus of this research is the development of a program that processes application software written in Ada, inserting self checking constructs into the overall software package. The self checking constructs are designed to detect single event upsets that cause deviations in program flow.

This technique is particularly targeted for microprocessor-based satellite control systems that have weight and power limitations imposed upon them. For this reason, it is important that any upset detection technique not incur extensive, if any, hardware overhead.

2.0 INCORPORATION OF UPSET DETECTION TECHNIQUES

During execution of real-time software, a processor continuously executes a defined software block. A transient fault in a control register of a processor may cause the program flow to deviate from the normal path and resume execution in another block of software (Figure 1).

To detect this deviation, block checking constructs are inserted into the application software. This requires the application software to be defined in terms of blocks, with a block constituting a logically bound set of instructions around which the block checking modifications are to be inserted. The constructs consist of TAG initializations, TAG checking, and TAG resetting. The placement of the block checking constructs is shown in Figure 2.

To insure detection of deviated program flow, a hardware watchdog timer is also set upon correct entry to a block. A watchdog timer that is not reset before its terminal count is reached will signal an error. The watchdog timer will detect infinite loops and reduce error detection latency. Figure 3 illustrates the flow diagrams for both the unmodified and modified blocks.

2.1 Incorporation of the Block Checking Structure

This approach to inserting upset detection mechanisms in pre-existing Ada software requires block boundaries to be defined. Definition of these boundaries is not too difficult since Ada is a block structured language. The entities that are defined to be blocks are procedures, functions, and tasks. Each block is assigned a unique identifying value. Upon correct entry into a block, the ID of the block is assigned to a TAG variable. The TAG retains this ID value until the end of the block is reached. The TAG is then checked for the block ID value and, if correct, is reset to the value zero. If the TAG does not contain the correct value, an error is signaled.

The insertion of the block checking constructs into software is divided into two parts. The first part is the initialization of the TAG; the second part is the resetting of the TAG. Additional checks are inserted if the body of the block is lengthy and/or loop structures are present.

This work has been partially sponsored by the Naval Research Laboratory under contract FE238311.

The internal structure of a block may contain code that is either executed once or many times, as in a loop structure. For long code blocks that do not contain loop structures, additional block checks are inserted after a certain number of statements to minimize latency in detecting faults following an incorrect block entry. For example, consider a program unit in which the TAG value is checked only at the entry and exit points. If an SEU causes a program flow deviation into this block past the TAG initialization at the start, upset detection will not occur until the TAG check at the end of the block. Depending on the block length and point of entry, a long latency period may occur. The insertion of a TAG check at some point within that block would minimize the detection latency period.

Code blocks that contain loop structures present another possibility of a long detection latency period. In Ada, loop structures occur in three forms: basic loops, while loops, and for loops. The ***basic loop*** allows a sequence of statements to be executed until a certain condition is met. The statements of a ***while loop*** are executed while a certain condition remains TRUE, and a ***for loop*** is executed once for each value in the discrete range of the control variable. Both of the latter loop structures may contain an exit statement to allow a premature exit.

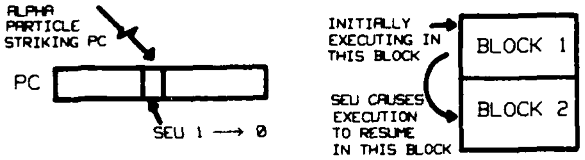

ALPHA PARTICLE STRIKING PC
PC
SEU 1 ⟶ 0
INITIALLY EXECUTING IN THIS BLOCK
BLOCK 1
SEU CAUSES EXECUTION TO RESUME IN THIS BLOCK
BLOCK 2

Fig. 1. Effect of Single Event Upset

Block checks may be included within the loop structure to minimize upset detection latency. If an SEU causes execution to resume within a loop structure not containing any TAG checks, a long detection latency period may follow, depending upon the length of the loop structure and the number of iterations to be performed. Inserting a block check within the loop structure will prevent the statements from being executed more than once before the upset deviation is detected. The additional memory and execution time overhead will be of greater proportion for loops that contain a few lines than for loops that contain a much greater number of statements. Since there is no way of always determining the value of a loop control variable, a TAG check should always be incorporated into the loop structure when possible.

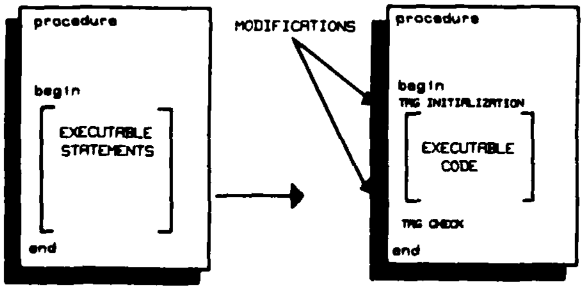


Fig. 2. Placement of Block Checking Constructs

2.2 Watchdog Timer Mechanism

Although the insertion of block checking constructs into application software is an effective means of detecting deviations from normal program flow, some deviations will go undetected by this method. These deviations cause program flow to resume in the same block while skipping statements.

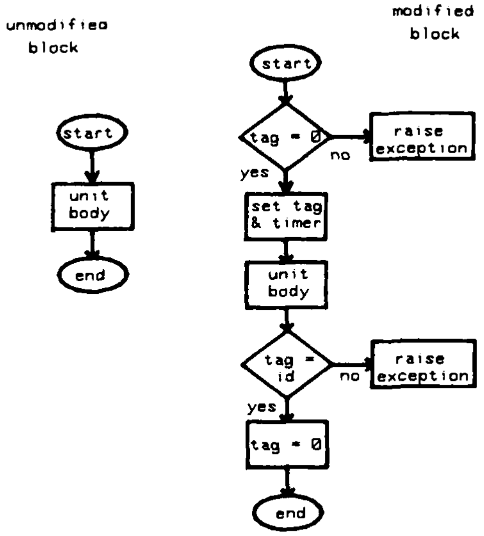


Fig. 3. Block Modification

If program flow resumes within a loop structure, certain loop variables may fail to be initialized, possibly resulting in an infinite loop. A hardware watchdog timer is a simple and inexpensive means of detecting these infinite loops and reducing upset detection latency. The timers are loaded with values at pre-established checkpoints in the software. The timer counts down and is reset before the next checkpoint is reached. If the timer is not reset before it expires, the corresponding process has probably failed in some way and the timer signals the processor of the error.

Implementing a block checking structure and watchdog timer mechanism requires that a programmer explicitly incorporate the upset detection mechanisms into the software at the time of development. The next section discusses the program that was developed to automatically insert the block checking constructs and watchdog timer mechanisms into Ada application software.

3.0 SOFTWARE MODIFIER FOR UPSET DETECTION (SMUD)

An automated approach to incorporate upset detection mechanisms into software would free the programmer from the task of having to do this manually. In addition, the automated approach requires that a programmer have only a superficial knowledge of self checking techniques. Automation also provides for uniformity of code. While incorporating a self checking structure manually allows each software system to be tailored individually, large software systems would benefit from the development time savings produced by an automated modifying procedure.

A program to pre-process Ada application software has been developed to automatically insert block checking constructs and a watchdog timer mechanism into the software. structure of the application software. The program, Software Modifier for Upset Detection (SMUD), has been developed using Ada and is designed to modify Ada application software. Directives are available that instruct SMUD as to the placement of these detection mechanisms. These directives allow a programmer to tailor a program unit to omit or insert additional statements into the code. SMUD can be modified to handle other high level languages that are block structured. The modifications mainly involve routines affected by keywords defined by the particular language.

3.1 Modification Procedure

The procedure of inserting block checking constructs into existing software relies on the use of certain keywords defined in Ada. Since the executable body of each program unit is bounded by the keywords ***begin*** and ***end***, a block boundary is easily recognized. When either of these keywords is encountered, a TAG initialization construct (after keyword ***begin***) and TAG reset construct (before keyword ***end***) is inserted into the source file. The instruction for initialization of a watchdog timer is placed after the initialization of the TAG.

As shown in Figure 4, SMUD processes the application software and inserts, the upset detection mechanisms.

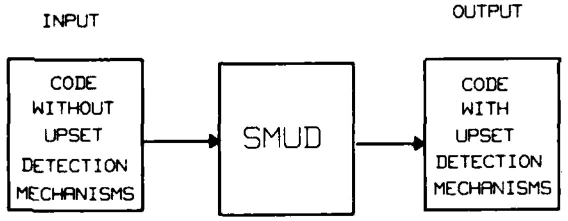

INPUT
OUTPUT
CODE WITHOUT UPSET DETECTION MECHANISMS
SMUD
CODE WITH UPSET DETECTION MECHANISMS

Fig. 4. Block Diagram of SMUD

To keep the original software intact, the modified version is output to a different file, as shown below:

```
        file1.ada --> SMUD --> UDMfile1.ada ──┐
   executable self checking code <-- compiler <┘
```

The modification procedure requires that the application software be partitioned into ***virtual nodes***. A ***virtual node*** is defined as a set of program units that execute on the same processor. Assuming that the application software to be modified has been partitioned into virtual nodes by the programmer, SMUD first prompts for the number of nodes contained within the entire system. SMUD processes each program unit contained within a node before processing another node. The modification process consists of two passes (PASS_ONE and PASS_TWO) through each program unit. This is illustrated in Figure 5. A brief description of the modification process for a virtual node is described. Listed below is the interactive information required from the programmer for each node:

1. Names of program units contained within the node
2. Physical memory mapped address of watchdog timers
3. The base in which the timer address is represented (i.e. binary, hexadecimal, etc.)

Procedure PASS_ONE is called to gather information about each program unit. Each program unit is assigned a record structure that contains the following information:

```
type check is (place, show,omit);

type program_unit is
  record
    kind : string (1..9);              --program unit type
    name : string (1..max_length);     --name of unit
    len : integer;                     --string length of name
    id : integer := 0;                 --block ID
    num_lines : integer := 0;          --no. executable statements
    watchdog : integer := 0;           --watchdog timer ID
    block : check;                     --type indicates option
    WDT : check;                       --type indicates option
    TAG_name : string(1..4);           --TAG variable name
  end record;
```

After PASS_ONE has processed each program unit and recorded the information, PASS_TWO is then called to reprocess each program unit. Procedure PASS_TWO implements the upset detection mechanisms according to the information obtained in procedure PASS_ONE. The information stored in the record structure of each program unit is examined before insertion of any block checking constructs or timer activation instructions. PASS_ONE merely gathers information about the program units whereas PASS_TWO actually modifies them.

The number of watchdog timers necessary for each node is defined by the number of program units that may run concurrently on a single processor. In addition to the main program, there may be tasks defined within the node. Since these tasks may run asynchronously, each task requires a watchdog timer different from those used by other tasks. If separate timers are not provided for each task 'he timer being accessed simultaneously by the tasks will be reset at improper times. Each procedure and function contained within the scope of a task unit or the main procedure accesses the same watchdog timer as its parent task unit. This configuration does not present a problem of conflicting settings of watchdog timers since each parent task unit may be

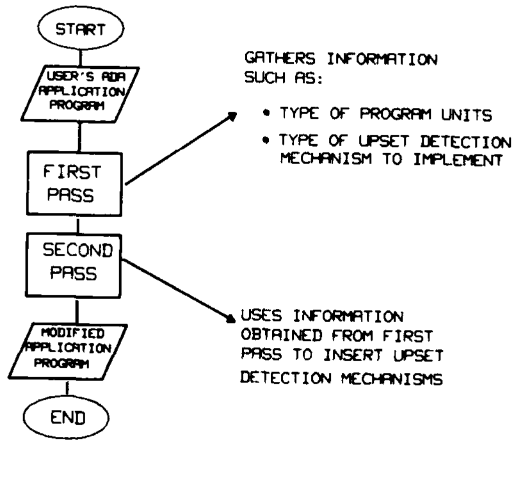

START
USER'S ADA APPLICATION PROGRAM
FIRST PASS
SECOND PASS
MODIFIED APPLICATION PROGRAM
END
GATHERS INFORMATION SUCH AS:
• TYPE OF PROGRAM UNITS
• TYPE OF UPSET DETECTION MECHANISM TO IMPLEMENT
USES INFORMATION OBTAINED FROM FIRST PASS TO INSERT UPSET DETECTION MECHANISMS

Fig. 5. Modification Process

viewed as a process running sequentially executed program units.

Two additional files result from the processing of each node: ***Data.Ada*** and ***Upset_Detection.Ada***. These files are created after each program unit has been processed by PASS_ONE. The first file, ***Data.Ada***, provides documentation of the modifications in the application software. The data includes the block ID of each program unit and the directives that were encountered in the parsing of each unit.

The second file, ***Upset_Detection.Ada***, is a package unit that must be made accessible to the application software by using the ***with*** clause. The declaration for the watchdog timers and the user defined exception is placed within this package. The exception handler for the user-defined exception, INCORRECT_TAG, is also contained within the package body UPSET_DETECTION. It is by means of this exception handler that a recovery sequence may be initiated. For purposes of testing the upset detection mechanisms in the simulation, the handler displays an error message each time a deviation in program flow is detected.

3.2 Directives

A directive is defined as an instruction to SMUD concerning placement of the two upset detection mechanisms. The purpose of providing directives is to allow a programmer to dictate what modifications should occur in each program unit. These directives are placed in the application software prior to executing SMUD. When any of these directives are encountered within defined block boundaries during procedure PASS_ONE, the information is stored in the record file of the program unit. In the absence of any directives, the program will insert block checks and initialize watchdog timers in the unit according to the techniques outlined in the previous sections. A list of the available directives and their functions is given below:

omit_blk omit insertion of block checking constructs in program unit

omit_wdt omit insertion of watchdog timer instructions in program unit

show_blk instructs SMUD to indicate locations where block checking constructs would appear. The locations are shown by commented statements that are inserted into the application software.

show_wdt instructs SMUD to indicate the default locations of placing the reset instruction of the watchdog timer.

blk instructs SMUD to insert a TAG check into the application software wherever this directive appears

wdt instructs SMUD to insert the instruction to reset a watchdog timer wherever this directive appears

omit_tag instructs SMUD to forego inserting a TAG check into a loop structure. This directive must be placed on the same line as the word ***loop***

4.0 SYSTEM MODEL

In order to test the effectiveness of SMUD, a system model has been developed (Figure 6). As previously mentioned, the system implements the MIL-STD-1553B communications protocol [12], which defines the modes of operation between satellite subsystems. The system model was developed in Ada with DEC's VAX 11/785 acting as the host.

4.1 The MIL-STD-1553B Communications Protocol

The MIL-STD-1553B defines a three layered communication protocol that includes the physical layer, the medium access control layer, and the logical link layer. The physical layer and the medium access control layer, which are modeled as a portion of the simulation environment define the interface between each subsystem and the 1553B bus. The logical link layer, which manages the protocol at a higher level, consists of the software that has been implemented in Ada.

Each subsystem shown in Figure 6 contains an application processor and an electronic module, called a BIU, that interfaces a serial data bus to the subsystem. Each BIU in the system can take on one of two roles: Bus Controller (BC) or Remote Terminal (RT). A third role for the BIU is described in the original protocol definition as a Bus Monitor. This module is not simulated since active information transfers only occur between BCs and RTs while a Bus Monitor receives selected data to be used at a later time. A BC is a system component that is responsible for initiation of any information transfer. An RT is incapable of initiating information transfers and responds to command words sent out by the BC. Each RT responds to its own unique address as well as a common broadcast address.

There are four types of information transfers that are used in the communication protocol operation [12]:

1. Bus Controller to Remote Terminal (BC to RT)
2. Remote Terminal to Bus Controller (RT to BC)
3. Remote Terminal to Remote Terminal (RT to RT)
4. Mode Command

Mode commands are used in managing the information flow of the communication system and detecting possible errors having occurred in the form of data corruption. There are three types of words that are transferred during information transactions: command words, status words, and data words. The command word defines what type of information transfer is to occur. The status word is transmitted from each RT during an information transfer unless an error occurred during the transfer or the command was sent in broadcast mode. The three data transfer types allow data words to be transferred between the BC and an RT or between any two RTs.

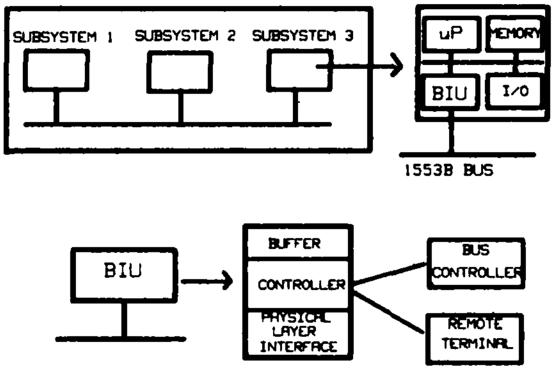


Fig. 6. System Model

In a BC to RT data transfer, the BC sends out a command containing an RT's address (or the binary address 11111 in the case of a broadcast command) and the number of data words to be transferred. Once all the data words have been transferred, the receiving RT sends the BC its status word (except in the broadcast mode). The BC verifies the status word to determine whether or not any data corruption has occurred.

The RT to BC data transfer follows the same procedure as the BC to RT data transfer. Once the BC issues a command indicating that an RT to BC data transfer should occur, the BC first waits for the transmitting RT to send back its status word. After sending its status word, the RT proceeds to send the required number of data words.

In an RT to RT data transfer, the BC sends out two command words. The first command word indicates which RT will be receiving the data while the second indicates which RT will be transmitting the data. The BC waits for the transmitting RT to send its status word. After the correct number of words have been transferred, the receiving RT sends the BC its status word where it is checked to determine the success of the information transfer.

4.2 Simulation Environment

The simulation environment has been developed in Ada using packages to encapsulate the components of the subsystem. Each subsystem consists of an application processor, a BIU, a DMA controller, and a host memory (Figure 6). These components are modeled at the functional level using task units, with the exception of the host memory which is modeled as a 2-D array. The BIU consists of a memory buffer, a controller, and a physical layer interface. The memory buffer is modeled as an array of words with each word consisting of 16 bits (elements).

4.3 Software Model

SMUD modifies that part of the simulation environment which is responsible for exercising the 1553B protocol. The protocol software resides in the controller element which forms a part of the BIU. The protocol model consists of a main tasking unit and fourteen subroutines. Each subsystem is identical in code, with the exception of the BIU ***addresses*** and the application processor software.

Each BIU is assigned a unique address, (e.g. BIU_1 is assigned the binary address 00001 while BIU_2 and BIU_3 have the binary addresses 00010 and 00011, respectively). This software model can easily be extended to include up to 31 BIUs (addresses 0-30). Binary address 11111 is reserved for the BC to communicate in broadcast mode with selected RTs simultaneously. Since each BIU may have the capability of being either a bus controller or a remote terminal, two boolean flags that indicate its current configuration are incorporated. The two flags, BUS_CONTROLLER_MODE and REMOTE_TERMINAL_MODE, are initialized in the declarative section of each BIU model. Only one BIU model has BUS_CONTROLLER_MODE assigned the value of TRUE at any given instant, while the other BIUs have REMOTE_TERMINAL_MODE assigned the value of TRUE.

Each task BIU has two procedures that are executed according to the configuration type: BUS_CONTROLLER and REMOTE_TERMINAL. For the BIU model that is designated the Bus Controller, the procedure BUS_CONTROLLER_1 determines whether the information transfer type is a mode command or a data transfer. Another procedure named MODE_COMMAND is called to further decode the command word. There are two types of mode commands: those that have an associated data word and those that do not. A data word that needs to be received/sent from/to the remote BIU is stored or read from the buffer. If the information transfer is a data transfer, the procedure DATA_TRANSFER is called. The command word passed into the procedure is checked for the type of data transfer and calls an appropriate procedure to carry out the data transfer.

Each BIU that is configured as an RT calls the procedure REMOTE_TERMINAL to determine whether or not it should respond to the data on the bus lines. If the data is a command word, the RT whose address is indicated in the command word further decodes the word. One of two procedures, PROCESS MODE COMMAND or PROCESS DATA TRANSFER is called to complete the required information transfer.

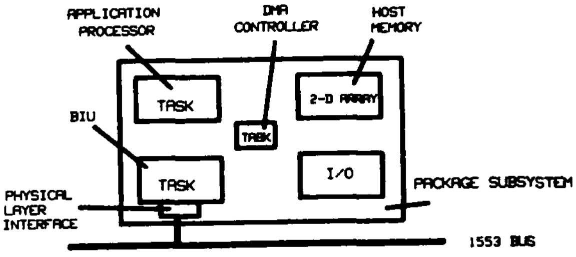


Fig. 7. Components of Simulation Environment

5.0 TESTING AND RESULTS

This section will discuss time and memory overheads incurred by the incorporation of a block checking structure and watchdog timer mechanism. The testing methodology and performance measures are presented here.

5.1 Testing

Testing of the upset detection mechanisms involved simulating the effects of SEUs causing a deviation from normal program flow. To simulate this type of upset, the application model containing the detection mechanisms was once again altered. This simulation consisted of modifying the application software to contain several entry points (Figure 8). These entry points would be activated to simulate the result of an erroneous jump by the program counter. The following sample program serves to illustrate how each procedure and function in the application software was altered for testing purposes. The sample program consists of four output statements to show the execution flow.

The output shown below is the result of (a) and (b) and shows the proper output sequence for a correctly executed set of instructions.

```
statement 1
statement 2
statement 3
statement 4
```

The modified test code will output the same sequence of statements provided each entry point is accessed successively. A driver routine was required to invoke the entry calls of the modified block. In the application model, every program unit behaves as a driver to each program unit contained within its scope.

(a)

```
procedure test is

TAG1 : integer := 0;

begin

 if TAG1 = 0 then
   TAG1 := 1;
 else
   raise INCORRECT_TAG;
 end if;
 WDT1 := 255;
 PUT_LINE("statement 1");
 PUT_LINE("statement 2");
 PUT_LINE("statement 3");
 PUT_LINE("statement 4");
 if TAG1 = 1 then
   TAG1 := 0;
 else
   raise INCORRECT_TAG;
 end if;

end test;
```

(b)

```
task body test is

 TAG : integer := 0;

begin

 loop
  select
    accept ONE;
    if TAG1 = 0 then
      TAG1 := 1;
    else
      raise INCORRECT_TAG;
    end if;
    WDT1 := 255;
    PUT_LINE("statement 1");
  or accept TWO;
    PUT_LINE("statement 2");
  or accept THREE;
    PUT_LINE("statement 3");
  or accept FOUR;
    PUT_LINE("statement 4");
    if TAG1 = 1 then
      TAG1 := 0;
    else
      raise INCORRECT_TAG;
    end if;
    exit;
  end select;
 end loop;

end test;
```

Fig. 8. Modifications for Testing Purposes

The driver routine shown below accesses each entry point in succession. The output resulting from the execution is identical to that of the original test program.

```
procedure driver is

 begin

  TEST.ONE;
  TEST.TWO;
  TEST.THREE;
  TEST.FOUR;

 end;
```

Since the altered software provides the means for selection of any four output statements, a program flow deviation can be simulated by eliminating an entry call and successively invoking the remaining entry calls. In order to simulate an SEU in the program counter, the driver routine bypasses the first entry point. This causes omission of the TAG initialization. As a further example, consider the driver routine shown below:

```
procedure driver is    Output from test program

 begin                 statement 3
                       statement 4
  TEST.THREE;          incorrect block entry
  TEST.FOUR;
end;
```

Upon reaching the TAG check at the end of the test program unit, the TAG is checked for the identifying value of the block. Since the TAG was not reset at the start, the value is incorrect and the exception INCORRECT_TAG is raised. The exception handler displays a statement indicating an incorrect block entry has occurred. Similar modifications have been incorporated into each subprogram unit of the application software.

The previous example served to illustrate a program flow deviation into another block. However, SEUs that cause execution to continue in the same block, though skipping several instructions, will not be detected by the block checking structure. For example, consider the following driver sequence:

```
procedure driver is

 begin                 Output from test program

  TEST.ONE;            statement 1
  TEST.FOUR;           statement 4

 end;
```

The driver accesses the first entry point, initializing the TAG to the identifying value of the block. The second and third entry points have been omitted, simulating a program flow deviation. Upon reaching the end of the block, the TAG is checked for the ID value. Since the TAG was initialized properly, the deviation is not detected.

Detection of this type of program flow deviation depends on the location where execution resumes in the block. As discussed earlier, a block containing a loop structure poses a potential infinite loop situation. A deviation near or into a loop structure may prevent the setting of necessary loop variables, thereby leading to the possibility of an infinite loop. The failure of the watchdog timer to be set would cause the timer to signal an error.

5.2 Results

The upset detection mechanisms yielded a 91 percent detection rate. The undetected errors were due to those program flow deviations that were contained within the block.

5.3 Time and Memory Overheads

The unmodified and modified system models were compiled using DEC's VAX Ada compiler. The memory overhead incurred by inserting the block checking constructs and watchdog timer mechanism into the application software model was 10.44 percent.

Two time overheads were obtained: the compilation time and the execution time. The compilation time represents a one-time overhead while the execution time overhead is incurred continuously as the modified application program is executed. The compilation time for the unmodified and modified application models was dependent on the load of the host computer. Similarly, the execution times varied according to the load and the non determinism of task selection. Further studies of the overhead is underway,and is reported in [13].

6.0 CONCLUSION

This paper has presented two techniques for detecting single event upsets that cause a deviation from normal program flow. These upset detection techniques consist of block checking constructs inserted into the application software and a hardware watchdog timer mechanism. A program has been written to automatically modify Ada application software to contain these mechanisms.

A system model that employs the 1553B bus communications protocol has been used to verify proper operation of the automated modification. The effectiveness of the upset detection techniques was determined through computer simulations.

REFERENCES

1. Elbert, T.F.,***Embedded Programming in Ada***, Van Nostrand Reinhold Company, New York, 1986.

2. United States Department of Defense, ***Reference Manual for the Ada Programming Language***, ANSI/MIL-STD-1815A-1983, February 17, 1983.

3. Tedd,M., Crespi-Reghizzi,S., and Natali,A.,***Ada for Multi-microprocessors***, Cambridge University Press, Great Britain, 1984.

4. Sosnowski, J., ***Transient Fault Effects in Microprocessor Controllers***, REL-CON EUROPE 86 Conference, Copenhagen, June 1986.

5. Oak, J., ***Block Checking Approach to Self-Testing Software***, MS Thesis EE, Virginia Tech, Sept. 1984.

6. Siewiorek, D. and Schwarz, R.,***The Theory and Practice of Reliable System Design***, Bedford, MA, Digital, 1982

7. .Shin, K. and Penz,D., ***Modeling of Concurrent Task Execution in a Distributed System for Real-Time Control***, IEEE Transactions on Computers,Vol.C-36, No.4, pp. 500-516, April 1987.

8. Stammers, R.A. ***Ada on Distributed Hardware***, Proceeding Workshop on Hardware Supported Implementation on Concurrent Language in Distributed Systems, 1985, Elsevier Science Publishers, pp. 35-40.

9. Randell,B., Lee.R.A., Treleaven, P.C., ***Reliability Issues in Computing system Design***, Computing Surveys, Vol. 10 no. 2, June 1978, pp. 123-165.

10. Rasmussen, R., ***Computing in the Presence of Soft Bit Errors***, Proceedings of Americans Control Conference, June 1984, pp. 1125-1130.

11. Booch, G. ***Software Engineering with Ada***, Benjamin Cummings Publishing Co., Menlo Park, CA, 1983.

12. Military Standard (MIL-STD-1553B), Aircraft Internal Time Division, Command/Response Multiplex Data Bus, Department of Defense, USA , 21 September 1978.

13. Heironimus, E., ***Automated Incorporation of Upset Detection Mechanisms in Distributed Ada Systems***,Master's Thesis in Preparation, Department of Electrical Engineering, Virginia Polytechnic Institute & State University, 1987.

Joseph G.Tront was born in Buffalo, NY, on October 20, 1950. He receive B.E.E. and M.S.E.E. degrees from the University of Dayton, Dayton, OH, in 1972 and 1973, respectively, and the Ph.D. degree in Electrical Engineering from the State University of New York at Buffalo, Amherst, NY, in 1978.

Since 1978 he has been a member of the faculty of the Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA. His research interests are in the areas of multiple-valued logic, microprocessor applications, computer aided analysis and digital electronics. In addition to his work within the University, he was also a consultant for McDonnell Douglas Astronautics, St. Louis, MO, where he has been involved in a study of the susceptibility of integrated circuits to electromagnetic interference.

Elisa K.Heironimus was born in Oklahoma City, Oklahoma on April 6, 1963. She received her B.S. and M.S degrees in Electrical Engineering in 1986 and 1988, respectively, from Virginia Polytechnic Institute and State University, Blacksburg, VA.

She is currently employed at TRW in Fairfax, VA. Her research interests lie in the areas of modeling and simulation and fault tolerant computing.

Research on the Ada Conversion of a Distributed, Fast Control Loop System

Eric N. Schacht

Computer Sciences Corporation

Abstract.

This paper discusses the experiences of a research effort to support the conversion of software written in PL/M-86 to Ada for a DoD missile system employing fiber optic technology. The hardware and software architecture of this embedded, distributed real-time (60 hertz control loop) system is examined, with emphasis on the special functional and performance requirements of this and other such applications. Ada programming techniques developed and benchmark tests conducted during this research effort are presented. The real-time programming techniques explained provide practical approaches to the implementation of Ada in an embedded, distributed real-time environment, and highlight certain critical capabilities required by an Ada compiler used in such environments. Results of the benchmark tests reveal the practical functional and performance concerns encountered by applications of this type, and further help to identify the vital qualities required by the supporting Ada compiler. Finally, Ada's ability to support the demanding requirements of a performance critical embedded application is evaluated in light of the experiences and lessons learned from this research effort. Practical suggestions are offered for compiler capabilities beyond the existing Ada language standard which are essential for the successful application of Ada to real-time distributed systems.

Architecture and environment overview.

The tactical gunner station computer for this weapon system provides various mission planning, communications, and navigational support for the vehicle in which it resides, and launch and flight control for a salvo of missiles. This gunner station computer consists of multiple, intercommunicating Intel 80186 microprocessors. These microprocessors communicate with one another over an IEEE-796 multibus. Each processor in the system can read or write into every other processor's dual port random access memory. The applications software is primarily written in PL/M-86. There is a small amount of ASM86 code used to manage communications over a fiber optic data link. No operating system is used.

One of the system's microprocessors is called the executive processor. It manages all communication between and data processing for the computer and gunner station console, digital area correlator, autotracker, land navigator, digital map generator, on-board sensor, and fire control processor. Other processors in the system are called autopilot processors - there is one for each missile which is in flight at the same time. The executive and the autopilot processors have the same computational capabilities but different input/output and memory requirements.

The gunner station computer must meet demanding performance requirements. It must accomodate a 60 hertz data transmission rate from the missile to the computer and back to the missile, and to and from an autotracker. Each autopilot processor must in 16.667 milliseconds receive 35 bytes of data from the missile, perform all guidance calculations, and return 15 bytes of data to the missile. The executive processor must in 16.667 milliseconds receive from, process, and send data to an autotracker, while simultaneously receiving from, processing, and sending data to the gunner's console, digital area correlator, fire control processor, land navigator, and digital map generator.

Techniques for communication across processors.

In the current PL/M-86 based system, the applications software is partitioned to run on a multiprocessor system based on a shared system bus. The microprocessors in the system interact with one another over the system bus through the use of data structures residing in shared memory. These data structures may have a local CPU address and a bus address for access by other CPUs. The local CPU addresses of the shared data structures are kept in pointers which reside at fixed, predetermined locations. If a program executing on one CPU has a need to communicate with a program running on another CPU through a particular data structure, then it is programmed with the knowledge of where the pointer to this data structure resides in physical memory. Strict memory and speed constraints drive the need for this kind of strategy where different applications running across multiple processors communicate through common memory structures. This is a typical architecture for a distributed, embedded real-time process control system.

There is another reason for this kind of programming strategy. During software development, the starting addresses of data structures used for inter-processor communications often change simply because of changes to the evolving program itself. But for the purposes of other programs on other processors needing to reference such a data structure, only the value of the pointer to the data structure needs to change. The advantage of this kind of programming practice is that changes to data structure size, composition, and starting addresses may occur without necessarily affecting, and thus forcing recompilation of, other programs dependent on these structures. Again, this is a typical kind of programming practice for a distributed, embedded system.

The question now arises as to how to provide interprocessor communications in an Ada conversion of this system. Several factors are at work which will influence the outcome of the final approach taken.

Consider the factor of extremely demanding real-world cost and schedule constraints for a possible Ada conversion. This pressure would dictate that the existing hardware and software scheme used for the gunner station computer system be preserved as much as possible. A significant and costly re-design effort would be required to make the system conform to a scheme which embodies practices associated with "good Ada" design.

The Ada purist would likely argue for the elimination of the use of common data structures on the grounds that it is bad software engineering practice. A pure Ada approach might advocate the use of interprocessor task communications as an alternative which offers a more sound design approach. But at the time of this writing, there is not exactly a plethora of validated Ada compilers which provide tasking across a configuration of bare Intel microprocessors. The same is true for compilers that support pragma SHARED in a distributed environment. This is a pragma which can be used as a means to synchronize the reading or updating of a variable. Another practical concern is that pragma SHARED can be applied only to objects whose type is a scalar or access type. Most of the current system shared data structures used in interprocessor communications have heterogeneous data elements, i.e., they are record type structures.

Given the current state of the world of Ada compilers targeting to bare Intel microprocessors, this research effort focused on developing Ada methods which parallel the current strategy for interprocessor communication. The Ada compiler used in the research to develop these methods did not support address clauses, which is not at all an uncommon characteristic for a lower cost Ada compiler.

Given this framework as a starting point, two basic questions must be addressed. These are:

* How can pointers be established at fixed addresses without the use of address clauses?

* Once a pointer is established at a fixed address, how do you access the data structure it points to from a program running on another processor?

To solve the first problem, the following seven step method was developed to declare an Ada access type object at an absolute memory location, and to link it to a data structure of interest.

Step 1. Define an access type which designates (points to) the type of the data structure of interest. (Refer to this as a level 2 pointer type).

Step 2. Define an access type which designates (points to) an object of the access type defined in step 1. (Refer to this as a level 1 pointer type).

Step 3. Instantiate the generic function UNCHECKED_CONVERSION to allow placement of the level 2 pointer at an absolute address by assigning an address value contained in an object of type LONG_INTEGER to the level 1 pointer.

Step 4. Instantiate the generic function UNCHECKED_CONVERSION to allow assignment of the address of the data structure of interest to the level 2 pointer (this can be overloaded with the above).

Step 5. Assign the absolute location value for the level 2 pointer to an object of type LONG_INTEGER. For the gunner station software conversion, this value must be an Intel segment:offset value equivalent to the absolute address.

Step 6. Declare the level 1 pointer object, assigning it the address value above using the instantiated type conversion function. The effect of this object declaration is to locate the level 2 pointer at the specified address. The object of interest, regardless of type, may now be addressed through this pointer.

Step 7. Link the level 2 pointer to the data structure of interest by assigning it the address of the data structure using the type conversion function as in step 6. The level 2 pointer now resides at the desired memory location, and points to the data structure of interest.

The following Ada program code demonstrates the use of this technique. An access type object DNLINK_ADDR_PTR is declared which positions a level 2 pointer at an absolute memory location. This object is used to access (point to) another common memory area called DOWNLINK_TABLE of type DOWNLINK_STRUCTURE, the ultimate data structure of interest.

```
-- PL/M-80 statement:

DECLARE DNLINK_ADDR_PTR AT (01E4H);


-- Ada equivalent:

type DNLINK_LEV2 is access DOWNLINK_STRUCTURE;
type DNLINK_LEV1 is access DNLINK_LEV2;
function CONVERT_ACCRESS_VALUE is new
    UNCHECKED_CONVERSION(SOURCE => LONG_INTEGER,
                         TARGET => DNLINK_LEV1);
function CONVERT_ADDRESS_VALUE is new
    UNCHECKED_CONVERSION(SOURCE => ADDRESS,
                         TARGET => DNLINK_LEV2);
ADDR: LONG_INTEGER := 16#0000_01E4#;
DNLINK_ADDR_PTR : DNLINK_LEV1 := CONVERT_ADDRESS_VALUE(ADDR);
```

Note that the above statements may be used in the declarative part of a subprogram body or in a package specification. The final step is to link the level 2 pointer to the target data structure. This can be accomplished as seen below:

```
DOWN_LINK_ADDR_PTR.all :=
    CONVERT_ADDRESS_VALUE
    (DOWNLINK_TABLE'ADDRESS);
```

Now the question arises as to how to reference the pointer at the fixed location and then reference the data structure it points to from another processor. The method used by the current PL/M-86 based system to accomplish this is described below. Next, an Ada technique is presented which parallels the method used by the current system.

To explain how public data structures located on other processor boards across the system bus are referenced, one of the autopilot processors will be used as an example. This processor's 32K bytes of dual port memory space resides in system memory space 60000H to 67FFFH. 60000H is the base address. If the executive processor needs to reference data from the autopilot processor board, it can access this system memory space to obtain it. But to the autopilot processor itself, this same system memory is local memory with an address space range of 00000H to 07FFFH.

As previously described, to enable message passing between processes, common data structures are used. Programs across all processors are built with the knowledge of where pointers to these structures reside in a particular processor's local memory space. At system initialization, the program which declares the pointer at a fixed address places the address value of the structure it points to within the pointer. This address value is the local address value with respect to the processor the program is running on.

Assume the software on the autopilot processor declares a pointer at 01D0H. From the standpoint of the executive processor, this same pointer is located at the absolute address of 601D0H. Thus, both the console processor software and the autopilot processor software know where the pointer is located. The autopilot processor software enters the pointer value at this location. This value corresponds to the autopilot processor local address space.

The executive processor software is now able to access the data structure of interest on the autopilot processor's memory space through the following process. First, it declares a pointer which resides at the system address of 601D0H. It then retrieves the value at this location, which is the local starting address of the data structure of interest. Next, it converts this value to a system address by using various PL/M-86 built-in functions to obtain the segment:offset values (Intel format), adding the system base value (60000H) for the autopilot processor to the local base value, and then reconstructing the new segment:offset values into the final system address value. It is now able to access the data structure on the autopilot processor.

An Ada technique for a program on one processor to obtain access to the memory space of a program running on another processor would basically follow the same process as described above. Using the Ada technique described above for declaration of an access type object at a predetermined location, an access type object could be "overlayed" on the local pointer, since its system address is known by the program seeking access to a data area. The value within this access type object could be retrieved, and adjusted as described above to arrive at the system address of the data structure of interest. With this information, the data structure may now be accessed through an access type object which contains this system address. It is the application of the generic function UNCHECKED_CONVERSION which makes this possible.

Now that a means to reference common data areas across processors has been established, the problem remains of ensuring that processes competing for access to that area do not cause corruption of that area due to simultaneous access. The pragma SHARED can be used to solve this problem. If pragma SHARED is not supported, as was the case with the compiler used in the research, then a standard approach which relies on a hardware supported test and set mechanism can be used.

PL/M-86 provides the LOCKSET function for shared memory synchronization between processes. It is used to test the value of, install, or remove a software lock for a shared memory location used as a flag which permits or denies access to a shared system resource. This function is implemented through an assert bus lock instruction which causes the microprocessor to assert its bus lock signal for the duration of the operation (i.e., testing of the lock value and installation or removal of the lock). When any external hardware receives this signal, bus access by other bus masters is prohibited as long as the signal is asserted. In this way, controlled access to shared resources can be enforced. For the research effort, an interfaced assembly language program was developed which provided the functional equivalent of the LOCKSET function.

Benchmark tests of floating point arithmetic.

Each autopilot processor executes a program which performs a lengthy series of floating point arithmetic equations to provide the function of missile guidance. Equivalent PL/M-86 and Ada benchmark programs were developed which contained numerous floating point arithmetic equations from this program. A test with runtime error checking enabled for the Ada program showed that the PL/M-86 program was 74% faster than the Ada program. This clearly demonstrated that runtime error checking is not feasible in the final system.

With runtime error checking suppressed, test results showed that the PL/M-86 program was approximately 18% faster. Analysis of the object code produced by both compilers showed that the PL/M-86 compiler used short real (32 bits) representation for real number data and instructions, whereas the Ada compiler used long real representation (64 bits). The compiler used for research did not allow control over real number representation. To help identify the cause of the performance difference, the compiler vendor ran a similar benchmark program using both the compiler without real number representation control and an under-development compiler which does provide real number representation control. The performance difference between the program compiled using short real representation versus the program compiled using long real representation was approximately 18%. This verified that real number representation differences accounted for the performance difference.

It is important to note that an 18% performance degradation would be considered intolerable given the current hardware configuration. Therefore, a compiler used in any future Ada conversion effort would have to offer control over real number representation.

A further note on these benchmark tests has to do with one equation which performed a square root calculation. As Ada provides no built in square root function and does not allow exponentiation using real numbers as exponents, a square root function using Newton's Method was developed. This function was found to be slightly more speed efficient than an Intel supplied library routine used by the PL/M-86 program for square root calculation.

Benchmark tests of shift and rotate and bit-wise Boolean operations.

There is a program module which processes autotracker data and runs on the executive processor. It contains numerous equations which perform shift and rotate and bit-wise Boolean operations. Various equations were taken from this module, rewritten in Ada, and tested for speed performance against the equivalent PL/M-86 program. Several approaches for providing these programming functions were tried until a satisfactory solution was found.

In the first approach tried, bit-wise logical operators from a special compiler specific package were used. Bit shift and rotate operators were built in Ada which used bit-masks to analyze operands bit-by-bit and then set the result in a bit-by-bit fashion. Benchmark tests showed this approach to be terribly inefficient, with the Ada programs being 30 - 50 times slower.

In the second approach, the Ada algorithms were rewritten so that shift and rotate operations were accomplished through multiplication or division by some power of two. Constraint checking must be suppressed for this approach to work. This approach improved the performance of the Ada benchmark programs significantly, but they were still orders of magnitude slower than their PL/M- 86 counterparts.

For the next test, the shift and rotate operators were built in interfaced assembly language programs. Shift and rotate function calls were linked to assembly language procedures. Benchmark test comparisons improved, but the Ada program was still 45% slower than its PL/M-86 counterpart.

Analysis of the object code produced by the Ada compiler versus the PL/M-86 compiler revealed the reasons for the performance difference. The PL/M-86 compiler generated inline instructions for the logical operators and the shift and rotate operators. The Ada compiler generated procedure calls to gain access to these functions. (The Ada compiler used does not support pragma INLINE).

Furthermore, the assembly procedure call generated by the Ada compiler was a FAR procedure call, rather than a more optimal NEAR procedure call. The research compiler does not allow control over the compilation model used, and an interfaced assembly language procedure must always carry a FAR label. When the ASM86 assembler encounters the control transfer instructions (JMP, CALL, RET, etc.), it uses the label of the called procedure (NEAR or FAR) to determine whether to produce an opcode that changes only IP (the instruction pointer register), or an opcode that changes both CS (the code segment register) and IP. This determines whether a one word or two word address is pushed on the stack when control is transferred to an interfaced assembly language procedure. The lesson from this experience is that for an Intel microprocessor based system, compilation model control is an important factor when trying to optimize an interface to an assembly language routine.

This 45% speed differential was still deemed unacceptable. Closer examination of the program module which processes autotracker data revealed that most of the equations using logical and shift and rotate operators were concentrated in mainly two areas of the program. To achieve an acceptable performance level, those parts of the equations from the two areas which use the shift and rotate and bit-wise Boolean operators were simply moved to two assembly language routines and interfaced to the Ada benchmark program. This approach brought the PL/M-86 vs. Ada performance differential down to 15%.

A new technique was then used to make parameter passing from the Ada program to the assembly language program as efficient as possible. All Ada objects to be manipulated by the assembly language program were arranged into a single record structure. The only parameter passed to the assembly language program was the starting address of this record. The interfaced assembly language program itself was built with the knowledge of this record structure. The speed efficiency of this approach is due to the fact that only the starting address of the record structure is pushed on the stack prior to the call of the assembly language procedure, rather than all of the Ada variables to be used in processing by the assembly language routine. This approach dropped the speed performance difference between the PL/M-86 and Ada test programs to only 5%. This difference is explained by the cost of the FAR procedure call.

Least significant (rightmost) bit tests.

There are numerous program statements in the missile guidance software which test the value of flag type variables before executing some process. PL/M-86 offers an efficient facility to do this through an IF statement which takes the form:

```
IF expression THEN statement-a;
ELSE statement-b; /* ELSE is optional */
   where statement-a is executed if the value of
   "expression" has a rightmost bit of 1.
```

A method was developed to provide an equivalent facility to the PL/M-86 IF statement using Ada. This method works by simply instantiating UNCHECKED_CONVERSION to transform a byte or a word type object to a Boolean object. The following code illustrates this method (the type BYTE was supplied by a compiler specific package).

```
with UNSIGNED; use UNSIGNED;
procedure IF_TEST is
A : BYTE : = 1;
function TO_BOOLEAN is new
      UNCHECKED_CONVERSION
       (BYTE,BOOLEAN);

begin
if TO_BOOLEAN(A) then
   ...
else
   ...
end if;
end IF_TEST;
```

When the TO_BOOLEAN function is used within an Ada IF statement as seen above, it performs exactly as the PL/M-86 IF statement does. If the operand passed to TO_BOOLEAN contains a rightmost bit with a value of 1, the THEN portion of the statement is executed. If the operand contains a rightmost bit with a value of 0, the ELSE part is executed.

The Ada compiler generated more efficient machine instructions than did the PL/M-86 compiler. Benchmark tests showed that the Ada program using this technique was about 40% more speed efficient than the PL/M-86 IF statement.

Lessons learned.

The experiences recounted in this paper confirm the shortcomings of Ada which are becoming more and more recognized in the embedded systems community. Very simply, the existing Ada language standard is not enough to ensure that an Ada implementation will unequivocally meet the special needs of an embedded real-time system. The needs of a distributed, embedded real-time system bring even further challenges, as this paper has shown.

There is a common theme which arises when analyzing the experiences of this research. All the experiences point to the need for easy to understand, easy to implement, and easy to maintain facilities for lower level representation and manipulation of system data elements.

The technique described to allow communication across processors is much more complex and "messy" than its PL/M-86 counterpart. The lack of a shared memory synchronization facility forced an assembly language procedure interface which does not exist in the current system.

The benchmark test of floating point arithmetic performance shows the importance of low level control over the internal representation of real numbers. The Ada language standard does not force an implementation to provide this level of control.

Severe performance degradation was experienced in benchmark tests of various Ada techniques used to perform bit shift and rotate operations and bit-wise Boolean operations. In many real-time embedded systems, numerous "black-boxes" communicate with a processor, sending and receiving data in various and inconsistent formats. These systems are required to transform and process this data in an efficient a manner as possible, and hopefully, the underlying programming language on which such systems are based will provide easy to use facilities to accomplish these tasks. The Ada language standard, however, does not require the provision of such critical facilities. Another factor surfaced in these tests is the need for compilation model control for an interfaced assembly language program (for an Intel environment). This is a "down in the dirt" level performance consideration, one which is not addressed by the Ada language standard. Nevertheless, it is a practical, real world need.

For a conversion effort, the luxury of a major system redesign to become more "Ada like" in character may not be practical due to hard cost and schedule constrairts, as is the case with this system. Likewise, existing investments in system hardware may prohibit upgrades to overcome performance deficiencies arising from the use of Ada. Therefore, solutions which parallel traditional programming approaches seen in embedded real-time systems are important.

When easy to understand, easy to use, and easy to maintain facilities are not available to meet the needs of a distributed, embedded real-time application, the life cycle costs of the system are adversely impacted. Maintainability falls in certain critical program areas. Vulnerability to errors when program changes occur is greater. The skill level in personnel developing and maintaining the system must be higher, and thus more costly.

The Ada implementation which is successful at meeting the challenging needs of distributed, embedded real-time environments must be based on a thorough understanding and appreciation of these needs. Moreover, it must provide more than is required by the current Ada language standard.

Eric N. Schacht is a Senior Computer Scientist in the Defense Systems Division of Computer Sciences Corporation in Huntsville, Alabama. His main areas of interest are software engineering, embedded real-time systems, Ada programming, and large scale project management.

Schacht received his BS in management science from the University of Alabama in Huntsville in 1976 and MS in computer science from DePaul University in 1983. He is currently pursuing a PhD in computer science at the University of Alabama in Huntsville.

Schacht's address is Computer Sciences Corporation, Defense Systems Division, 200 Sparkman Drive, Huntsville, AL 35805.

Implementing Distributed Ada* Tasking by Emulating the Rendezvous

by
James E. Tomayko, Minh Leo Pham, and Wang Wei,
The Wichita State University

and James E. Kroening,
Boeing Military Airplane Company

Abstract

Ada was designed to assist in the implementation of concurrency on distributed processors. To date, no effective compiler that supports Ada for distributed targets is available. One method of task distribution without direct compiler support is to translate an Ada tasking program into one that achieves the rendezvous using a communications package suitable for the target architecture. The use of this translator allows applications programmers to retain the advantages of developing a system using strong typing, data abstraction and information hiding, which are lost if the application is in the form of separate communicating programs. This paper presents the specification, design, and implementation of the Ada Rendezvous Emulation Tool (ARET) that accomplishes the translation and creates compilation units suitable for distribution on multiprocessors. The integration of the communications package with the tool, the difficulties encountered in emulating features of the rendezvous, and the effectiveness of the concept are also discussed.

Rationale for Emulation of the Ada Rendezvous

Embedded real-time software systems are being designed to run in a multiple processor environment, the environment envisioned by the designers of Ada tasking. The processors used in such systems often have small memories, in the 64 to 256 kiloword class. In the case of relatively large systems, this characteristic presents a serious problem in the distribution of software. A natural solution is to design the software in functional units implemented as tasks, with the tasks distributed on the processors and communicating via the rendezvous. Such designs would generally result in compilation units smaller than the available memories. Unfortunately, no off-the-shelf Ada compiler and associated run time system fully supports distributed tasking. Ada users who wish to achieve distribution are forced to develop application-specific operating system and run-time support for processes.

Most current ad hoc solutions to the distribution problem run counter to the intent of the design of the Ada language. Ada's features that help in data abstraction, information hiding, and strong typing are less effective when applied to a local program that is using some communications package to couple to a process on a different machine. Creating a distributable tasking program to implement an application helps promote good software engineering practice and the correct use of the language. In the absence of tools to fully implement distributed tasking and the rendezvous, a tool to take an Ada tasking application program and translate it into a collection of distributable processes that communicate through the emulation of the rendezvous can serve as an interim solution.

Models of the Distribution of Software

Background

The development of multiprocessor systems and the Ada(*) language both grew out of the improved understanding of the nature of software that began to have effect in the early 1970s. One original impetus for the development of multiprocessor systems was the idea of parallel execution. If a program could be separated into independent parts, or if a system could be functionally distributed, then having multiple processors available logically shortens the time it takes to complete a run, or increases the speed of a continuously executing system. An additional reason for multiprocessors is to support the concepts of modularity and data abstraction. Parnas' seminal work on decomposition, Shaw's interest in data abstraction, and Wulf's development of an experimental multiprocessor occurred in the same place and time and were mutually

*Ada is a registered trademark of the U.S. Department of Defense.

reinforcing. The software concepts and the hardware architecture go hand-in-hand.

Two types of multiprocessor architectures are presently dominant: shared memory and local memory. In the 1970s, Carnegie Mellon built one of each type. The shared memory system was called C.mmp and is documented in Wulf81. The local memory system was Cm* and is described in Jones78 and Siewiorek78. Both systems were built using LSI-11 or PDP-11 processors. (Ironically for those saddled with the 1750A, Wulf commented that his team would never have chosen a 16 bit machine in the first place, but that their commitment to off-the-shelf hardware precluded other choices in 1972.)

One result of the C.mmp experiment is the discovery that "large applications almost invariably want to address large amounts of data, even when they are decomposed for a multiprocessor." [Wulf81] This influenced Cm* somewhat. The local directly addressable space of 64K words on each processor node was augmented by a virtual memory system that enabled a processor to access other nodes in a cluster or other clusters. The Cm* designers added a microcoded routing device similar to a bus controller to speed up communication (the Kmap, see Figure 1). The relative time cost of accessing memory was 1:3:8 for local, intracluster, and intercluster communication. All memories were divided into 4K blocks, with addresses made using simple offsets and translation done by the routing device.

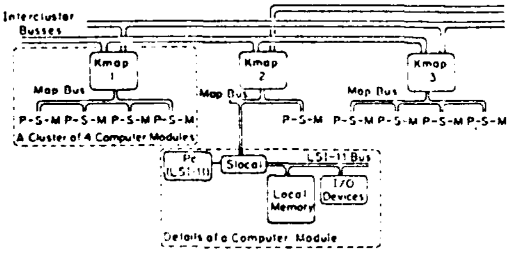


Figure 1: Cm* Architecture. Note the similarity to Pave Pillar-like architectures.

Cm*'s operating system, StarOS, supported the division of software into "task forces." User-defined relocation tables placed the tasks on the system. Unlike the later Ada tasks, StarOS did not block processes when messages were sent.

The concurrent specification and development of a design for Ada tried to exploit both the newly formalized software engineering principles and the possibilities of multiprocessors such as Cm*. It has been recognized that Cm* provided support for object oriented designs and software [Buzzard85]. Object oriented design methods facilitate, in turn, the partitioning of software for multiprocessing [Armitage85]. Ada has constructs that explicitly aid in both the implementation of object oriented designs and distribution on multiprocessors [DoD83, Booch86]. However, some feel that neither the syntax nor the semantics of the language properly define task distribution [Knight87].

The ideal model of task distribution is one task/one processor. Greater understanding of the concepts of coupling and cohesion in the design of modules and objects indicated that properly designed tasks would be able to achieve all needed communication through rendezvous, thus simplifying interprocessor coupling. Unfortunately, the requirement for a "parent" with a procedure name and at least a null statement in its body early eliminated the notion that all tasks were equal, and complicated the development of distributed tasking. Since distributed tasking is currently not supported by a validated compiler (though several efforts are underway, some promising results by the second quarter of 1988), other methods of distribution are being explored. Many of these methods are not limited to task constructs only.

Methods of Distributing Software

Previous experiences with distributed systems reveal a number of software distribution schemes, each of which has advantages and disadvantages. These are now evaluated assuming an architecture similar to that in Figure 1 and Ada software.

Separate programs: The most obvious method of distributing software on multiprocessors is to place a single, self-contained piece of software on each processor, and use standard message-passing schemes for inter-process communication. The advantage of this method is that it is simple to implement fault-tolerance. A spare processor, when loaded with another copy of the software, informed of critical data values, and assigned the logical address of the failed computer, completely replaces a faulty machine in the system. Another advantage is that exception handling is kept local to the processor.

The disadvantage of this method is that the hardware architecture forces the software design to be done in a highly constrained manner. It is much simpler to view a real time embedded system as a single program in order to make it possible to use abstraction correctly [Volz85]. Also, as the development life cycle progresses, it becomes increasingly difficult to reallocate functions when changes to the requirements occur [Armitage85]. Changes after initial operational capability have the potential to cause havoc. Because of these concerns, both GTE and Honeywell rejected this approach in their distribution systems [Armitage85; Eisenhauer86]. GTE develops and tests real time programs on the APSE first, then distributes the software. Honeywell emphasizes that the functional specification should be kept separate from the mapping to the underlying system. Portability is thus enhanced. It is better to treat the Ada software as a coherent whole, and postpone distribution until the software is functioning correctly.

Automated distribution: The first consideration in specifying an automated distribution system is the question of the granularity of the distributed units. Tasks seem like logical units, but "the Ada task is a construction for expressing concurrency and not modularity [Cornhill83]." Also, it has been found in practice that restricting the software to tasks only does not necessarily simplify the run time support necessary [Eisenhauer86].

Given that it is not essential to make the restriction to tasks, several options are available. A team at the University of Michigan uses library subprograms and library packages as the unit of distribution, with task objects never moving from the allocating unit [VolzFC]. Honeywell's system makes all units distributable [Eisenhauer86]. Both of these systems use a form of pre-translation to decompose the original Ada program. Michigan's translator looks for a pragma SITE in the units to determine where to place them. Honeywell's is more robust, with its own Ada-like language for program partitioning. GTE's is similarly powerful. Since these partitioning systems operate prior to compilation, it is possible to have heterogenous multiprocessors as a target. Also, exception handling can be artificially "localized" by the translation.

Access types: This method and the next one attempt to take into account the possibility of primary memory overruns. These may occur due to the inclusion of an excessive number of functions in the software, or due to reconfiguration after hardware failures.

Access types, specifically access task types, can be used as a method of implicit distribution. The run time system can place the new tasks on any processor. However, the newly instantiated tasks must be of the same type as the primary task, thus severely restricting the software developers [Knight87].

Dynamic Distribution: V. Santhanam suggested a method of distribution in which the minimum software necessary to instantiate the system is loaded on the processors, then procedure and function calls cause additional software to be loaded when needed.

To visualize this suggestion, imagine a large bushel of apples representing code units in non-volatile memory. A series of smaller baskets represent the individual processor memories. To begin, one apple is thrown from the big bushel into each smaller basket. When a procedure call, instantiation of a task, function call, package, or any other such unit is needed by a process, then it is loaded ("throw another apple into the basket"). If any individual basket gets full, then a needed apple is put into an emptier basket and required communications are set up to its parent process. If all the baskets fill, then the least needed apple is tossed out, and its place taken by the new apple. Virtual memory developers have determined algorithms for figuring out which software to swap.

This method is very powerful, but equally complex. It requires a total rethinking of the current distribution schemes. It also would require the development of a complicated run time system. On the positive side, it would provide robust fault tolerance capability.

Task distribution using simulated rendezvous: Given the absence of a compiler and run time system for distributed tasking, the Ada rendezvous can be simulated using more conventional message passing techniques. A communications package developed by co-author James E. Kroening for a system using the "separate programs" concept includes procedures that can be used as simulations of the entry call and accept statement in Ada. The availability of this package makes possible a translation tool that can take Ada source and replace the appropriate statements with the correct procedure calls, and include the communications package. The resulting code can then be compiled and distributed using the operating system allocation algorithms already in place.

A Communications Package for Emulating the Rendezvous

The approach implemented to provide communications between distributed processors running Ada applications is called Logical Addressing. A "single Ada program per processor" approach is used to partition the software, but the use of Logical Addressing is not limited to such an environment. The basic philosophy can be applied to a single program on a single CPU, a single program on distributed CPUs, or multiple programs on distributed CPUs. Discussion of the method, however, will be in the context of the single program per distributed processor environment.

Architecture Overview

The architecture for which the communications system is implemented is presented in Figure 2. Multiple processors are linked by a local data bus to form processor subsystems. One of the processors in each subsystem is a node on the system data bus. Communications between processors in the same subsystem can take place entirely over their local bus, but communications between processors in different subsystems involve at least one transaction over each of their respective local buses as well as at least one transaction over the system bus.

Processors on the local bus vie for temporary control of the bus, and direct memory access is employed for data transfer. Failures of single elements on either bus (local or system) do not cause failure of the entire bus.

Logical Addressees

Logical addressing is a method by which one Ada task can communicate with another (or even itself) without an awareness of the physical location of its communicating partner. The communicating tasks may be in the same processor, in different processors on a local data bus, or even in different processors in a more complex network. The task initiating the communications transaction will be referred to as the active side, and the remote or non-initiating partner will be referred to as the passive side.

The fundamental underlying concept to the Logical Addressing method is that of the logical addressee. The logical addressee is a system-unique identifier for a data area on the passive side of a transaction. When a task needs to get data from or send data to another task with which there is no guarantee of co-location on the same processor, it uses the logical addressee to specify the remote source or destination of the data transfer. The communications system is responsible for determination of the steps necessary to effect the data transfer.

Binding of the logical addressee value to a specific location can be done at load time or in a dynamic fashion at run time. Load time binding requires a component of the loader to be aware of the communications needs: the values of logical addressees associated with the code segment being loaded as well as the associated physical location. Run-time binding requires the declaration of the logical addressee value and its bound location by a task with visibility to the data object(s) to which the logical addressee is bound. Run-time binding, the method employed in this implementation, is more flexible in that logic can easily be applied to declare unique logical addressees for the same code in different circumstances. An example of the usefulness of this technique is when

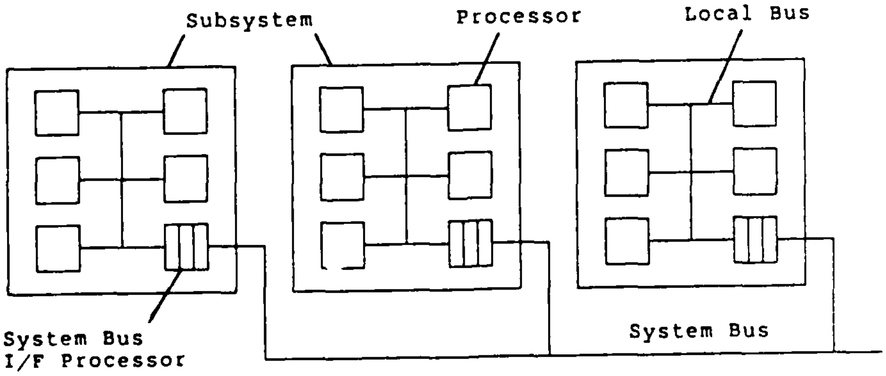


Figure 2: Basic System Architecture

common code is loaded in several processors with the need to communicate with each instantiation separately. Load time binding has the advantage of less requirements on the user at run-time to enable communications - a more automated approach.

The allocation of logical addressee values is done at design time in a manner that guarantees uniqueness of values used by different applications. Using integer values gives the capability of up to 65536 unique logical addressee values in a system (with a 16-bit integer), so that reasonably large ranges of values can be allocated to different communications users which will satisfy their needs without requiring unnecessarily detailed attention to the allocation.

When the communications system becomes aware of the emergence of new logical addressees (at load time or run-time), the appropriate information about the logical addressees is disseminated among the distributed databases. The information that is distributed allows the path of remotely-initiated messages to be determined in an efficient manner.

User Interface

The utilities provided by the communications system are classified as either active side or passive side services. The active side services provide a task the capability to initiate communication with some remote partner. Procedures PUT and GET allow a task to request the "push" or "pull" of data to or from a location designated by a logical addressee, and continue processing without waiting for the completion of the actual transaction. Procedures PUT_AND_WAIT and GET_AND_WAIT provide for the initiation of the same transactions while preventing further execution by the requesting task until the transaction has been completed (successfully or unsuccessfully).

The parameters required for active side service requests are:

- the logical addressee value designating the remote end of the transaction,
- the physical address of the local data area to be sent from or received to,
- the amount of data to be transferred,
- a transaction priority value, and
- the result flag.

The result parameter is passed by reference in the cases of PUT and GET, and by value in the other two cases. In this way, the communications service can update the flag when the transaction has completed for later consultation by the requesting task. The result flag is of an enumerated data type with values FAILED, SUCCESSFUL, PENDING, and LA_NOT_DECLARED (logical addressee not declared). The LA_NOT_DECLARED result is returned when the logical addressee value specified in a request is not known to the communications system.

Other active side services included in the specification but not implemented in this version of the communications system are GET_CYCLIC and PUT_CYCLIC. These procedures allow users to request the periodic initiation of a transaction without further intervention.

The passive side services provide tasks with the capability to bind logical addressee values to specific locations of data objects, and to control task execution based on the passive sending or receipt of data. Procedure DECLARE_LOGICAL_ADDRESSEE provides the binding function, and procedures SEND and RECEIVE provide the passive side task control.

Calls to DECLARE_LOGICAL_ADDRESSEE are usually made as tasks are initializing, but the timing of the DECLARE_LOGICAL_ADDRESSEE call can be used to inhibit transactions until some desired processing has completed. Once a logical addressee has been declared, it is bound to its associated location until the software responsible for its declaration is deallocated or moved for some reason. Parameters to a DECLARE_LOGICAL_ADDRESSEE call are:

- the logical addressee value
- the associated physical location, and
- the scope of the declaration.

The visibility of each declaration can be specified as PROCESSOR_ONLY, SUBSYSTEM_ONLY, or GLOBAL. It is left to the user's discretion to determine which is the most suitable scope for the declaration. If he or she has little idea of how software will be partitioned on the system, exclusive use of GLOBAL might be required. If, however, he or she is well aware of the partitioning for certain software elements, the use of PROCESSOR_ONLY or SUBSYSTEM_ONLY might simplify the software design (especially in the case of multiple instantiations of common code).

Procedures SEND and RECEIVE allow a task to delay further execution until at least one of the logical addressees passed as a parameter to the call is involved in a transaction. Up to ten logical addressees can be specified (this is an arbitrary limit), and the one that was involved in a transaction is returned as an output parameter. Also included is the capability to specify a timeout value, which is the amount of time after which the requesting task wants to continue execution, even if the logical addressee(s) specified are not involved in a transaction. This implementation of the communications system only remembers whether or not a logical addressee has been involved in a transaction, and not the number of transactions if more than one has occurred. Once a task is activated by virtue of a certain logical addressee having been involved in a transaction, no further tasks can be activated for the same logical addressee until another transaction with it has occurred.

Use of an active/passive pair like PUT_AND_WAIT and RECEIVE can accomplish not only a transfer of data but also a process synchronization. A common use of such a pair by other operating system functions in this implementation is as a semaphore.

Database Issues

The overriding concerns in the design of the communications system database are the conservation of memory and the maximization of speed. The philosophy adopted was that each processor's database would contain enough information to initiate the "next step" of an interprocessor transaction. Two types of databases were defined - the processor-level database and the subsystem-level database.
The processor-level database, resident in each processor, contains information about each logical addressee that has been declared in the subsystem with a scope of SUBSYSTEM_ONLY or GLOBAL. Additionally, information about each logical addressee declared as PROCESSOR_ONLY is contained only in the processor in which the logical addressee was declared. The information contained in the processor-level database is sufficient to fully complete transactions involving logical addressees in the subsystem. The processor-level database contains no information about logical addressees declared in other subsystems.

The subsystem-level database, resident only in processors having interfaces to the system bus, contains information about each logical addressee that has been declared in other subsystems with GLOBAL scope. The information contained for each such logical addressee is simply that which is required to forward a message across the system bus to the interface element of the subsystem on which the logical addressee resides.

Transaction Examples

When an active side request to initiate a transaction is made to the communications services, the processor-level database is consulted for information about the logical addressee specified in the request. If information is found, the transaction is initiated by performing a memory-to-memory move if the logical addressee is in the same processor as the active requestor, or by performing the I/O on the local bus otherwise.

If no information about the logical addressee is found in the processor-level database, a request block (and any associated data) is sent over the local bus to the interface processor on the system bus. The communications software in the interface processor consults the subsystem-level database to determine the subsystem location of the specified logical addressee. If no such information exists, then the logical addressee has not been declared to the system. and an appropriate message is sent back to the requesting processor so that the requesting task can be notified. Otherwise, the interface processor initiates a transaction on the system bus to the subsystem in which the logical addressee resides. Upon receipt of the system bus transaction by the interface processor on the remote subsystem, the resident communications software consults its processor-level database to determine the last leg of the outgoing transmission. Again, if no information is found about the logical addressee (due to a recent deallocation, for instance), an appropriate message is propagated back to the requesting processor. Otherwise, a local bus transaction is initiated on the remote subsystem, in which the required data is sent to or read from the processor in which the logical addressee was declared. Results (and data, if necessary) are then propagated back to the requesting processor over the same path as the outgoing request.

Fault-Tolerance

In order to ensure that an active side requestor always receives a result other than PENDING in a finite amount of time when inter-subsystem messages are initiated, a communications task monitors for the timely completion of the transaction. When no response is received after a certain amount of time, the requestor's result flag is set to FAILED. This sort of monitor philosophy is also employed at the system bus interface level, where, after certain timeouts, attempts are made to propagate failure messages back to the initiating processor.

As far as processor or subsystem failures are concerned, the focal points of fault-tolerance are the system bus interface processors in each subsystem. When a processor in a subsystem is starting or restarting for any reason, the interface processor in its subsystem is consulted for current processor-level database updates, so that the initializing processor can be aware of logical addressees declared in its absence. This implies that the interface processor itself must be fault-tolerant. Although this implementation of the communications system does not account for interface processor fault-tolerance (for cases in which the interface processor must restart), techniques exist for implementing reliable fault handling. Of course, hard failures of the interface processor will cause isolation of the subsystem in either case, but soft (recoverable) failures or power transients could be handled by employing techniques such as critical data buffering in a non-volatile memory module within the subsystem or on the interface processor.

The communications system depends on other parts of the operating system to inform it of the loss of some system component (a processor or entire subsystem) or of the deallocation of some software elements and their associated logical addressees. Once notified, the communications system disseminates the deallocation information across its distributed database, much as is done in the case of logical addressee declarations. Because of the database structure, loss (or deallocation) of a processor or subsystem only affects the subsystem-level databases in the remote system bus interface processors, and the processor-level databases in the processors in the subsystem on which the loss occurred.

Implementation Issues

An aspect of the architecture that had some influence on the design of the communications service was the type of local bus employed. Since the local bus has direct memory access capability and speed of transactions was of significant concern, an approach was developed to use direct memory access to read and write directly from and to the users' databases with the local bus. This has the advantages of saving large amounts of time and space that would be spent buffering data at each end of a transaction but introduces certain weaknesses to the overall communications system (which are discussed in a subsequent subsection), as well as certain limitations.

One limitation is that the granularity of the size of data passed by the communications system is dictated by that of the local bus itself. In this implementation, the bus works with 16-bit words (up to 4096 at a time), so that no data object smaller than one word can be separately moved by the communications system. Users are responsible for ensuring that objects required to be passed through the communications system are contained on word boundaries in their representations.

Another limitation is that non-contiguous data cannot be transferred in the same transaction. An implementation that buffers the data can collect it and then transfer it over the bus. Additionally, in the case of PUT, the communications system is unusable with dynamically declared data that might not be present a short time after the call to the communications service is made. Again, buffering the data when the request is made would prevent this limitation.

Expandability to Rendezvous-like Constructs

The Logical Addressing communications system provides services which could be used as primitives to build a system that provides rendezvous-like constructs for use between distributed tasks or distributed Ada programs. Implementations similar to McDermid's rendezvous methods [McDermid82] or Wellings distributed Ada program communication [Wellings84] are fairly well supported by the constructs provided in the current implementation. Adopting such a system includes incurring the penalty of the associated overhead, and a trade study analysis of the comparative virtues of different levels of implementation would have to be conducted to determine the best approach.

Requiring all inter-processor communications to use such constructs in an embedded system with time-critical operations could prove to be disastrous for certain applications. The aesthetic, maintainability, and correctness rewards to be gained are well-documented, but unless a system can meet its performance requirements such rewards are meaningless.

Major Weaknesses

Other than the limitations discussed in the Implementation Issues subsection, this communications approach contains at least two other weaknesses that merit discussion. The first and most significant weakness is that the Ada checking mechanisms are totally circumvented when data is moved directly to a database via direct memory access techniques. No type- or range-checking is conducted, so all data obtained through the communications system must be regarded as "unsafe data", and such security must be provided by the users. In military avionics mission computer systems, however, such direct memory access techniques are not only commonplace, but essential from a performance viewpoint.

Even though run time checking is disabled, compilation time checking still exists, so the integrity of the source code is maintained.

Another weakness is the violation of the Ada task data sharing principles. When the communications system asynchronously updates the result flag provided by a requestor (as can be the case with PUT and GET) or any data area designated in a request, the assumption stipulated by the Ada Reference Manual [DOD83] in section 9.11 that a shared variable is not updated by some task when another task reads it between synchronization points is violated. The effect of such an assumption allows Ada compiler implementations to retain shared variable values in processor registers between Ada synchronization points, creating a case in which a task may not realize when a result flag has been updated in memory. The communications system attempts to accommodate this possibility by the inclusion of a function to access result flags that guarantees access to memory for its evaluation. Elimination of the PUT and GET utilities (leaving PUT_AND_WAIT and GET_AND_WAIT) would alleviate the shared variable problems with result flags, but at the expense of causing tasks to wait for the completion of I/O in every request.

Major Strengths

The major strengths of the Logical Addressing approach to Ada program communication are as follows.

1. An off-the-shelf Ada compiler for a single program per processor implementation is all that is required to support the communications. This is especially important with the current lack of maturity in Ada compilers for embedded systems.

2. The system is reconfigurable and flexible to run-time changes in the processing environment. Future enhancements to embedded systems approaches such as task-level allocation and deallocation are supported by the communications approach.

3. The communications system specified meets the requirements given in the first section of this paper, and is thus useful in an embedded avionics environment.

The Ada Rendezvous Emulation Tool

We developed a tool to take advantage of the communications package features to implement distributed tasking. ARET, the Ada Rendezvous Emulation Tool, is actually a kind of preprocessor for Ada source code which replaces entry calls and accept statements in Ada source code by some specific procedure calls. These procedure calls can perform the rendezvous that the original entry calls and accept statements intended to do.

Fundamental considerations

What an Ada rendezvous does: Ada supports concurrent real time processing by its program units of tasks. The synchronization (rendezvous) between two tasks is achieved by a hand shake of an entry call and an accept statement in the two tasks respectively. Not only can the synchronization be achieved by means of the rendezvous, but also data transfer between the two concurrent running tasks. The parameters of the entry call and accept statement serve as the vehicle of the data transfer.

Purpose of ARET: Since up to now no effective compiler that supports Ada for distributed targets is available, a way of achieving distribution without direct compiler support is desired. ARET is designed to do this by:

> Eliminating all entry definitions, entry calls and accept statements from the Ada source code.
>
> Using calls of procedures from the communications package to do exactly what the eliminated entry calls and accept statements should do, that is, the rendezvous between two concurrent running tasks would be achieved by the procedure calls.

Critical Matters in Rendezvous Emulation: Two critical items were considered in designing the rendezvous emulation:

1. Means -- by what means can we achieve the emulation? A combination of calls of two procedures, PUT_AND_WAIT and RECEIVE, from the communications package is used to accomplish the emulation. Figures 3a and 3b show the working rationale.

2. Place -- where does the emulation rendezvous occur? An Ada rendezvous has an asymmetrical structure. That is, the task making entry call knows in which task the called entry is, while the called entry does not know which other task is calling it.

We do not normally care how an Ada compiler arranges for a rendezvous achieved by entry call and accept statements. However, when we want to replace entry call and accept statements by procedure calls to achieve a rendezvous, we have to create a place where the rendezvous can occur. The reason is that when a call of a procedure is made, the actual parameters are given

Figure 3a: The Original Rendezvous

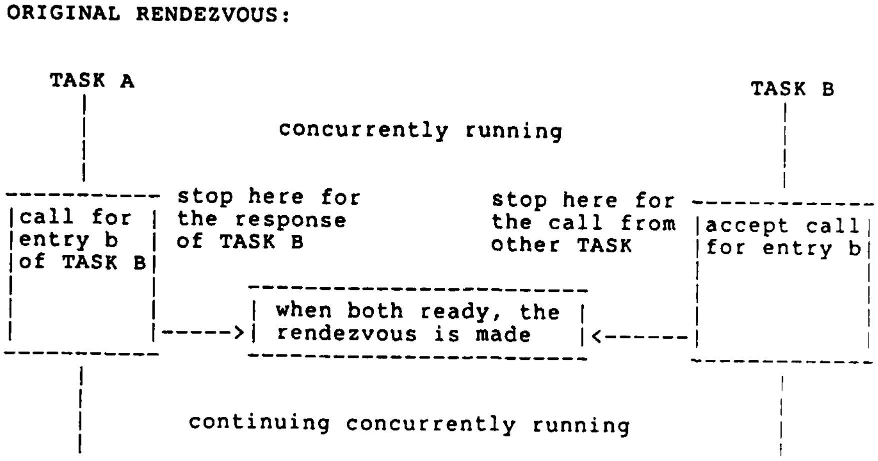
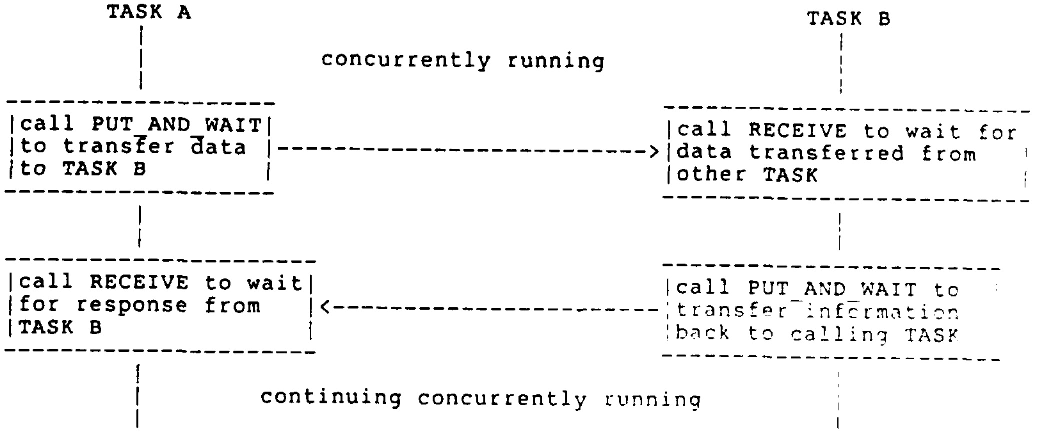

ORIGINAL RENDEZVOUS:
TASK A
TASK B
concurrently running
call for entry b of TASK B
stop here for the response of TASK B
stop here for the call from other TASK
accept call for entry b
when both ready, the rendezvous is made
continuing concurrently running

Figure 3b: The Emulated Rendezvous


TASK A
TASK B
concurrently running
call PUT_AND_WAIT to transfer data to TASK B
call RECEIVE to wait for data transferred from other TASK
call RECEIVE to wait for response from TASK B
call PUT_AND_WAIT to transfer information back to calling TASK
continuing concurrently running

explicitly by their names which also brings information about their types, their origins, and so on. Obviously, the asymmetrical structure of the rendezvous prevents us from using procedure calls to respond to another task's call directly. For that reason, we create a place called DATA_STATION. The procedure calls of PUT_AND_WAIT and RECEIVE from the calling task will send and get data and other rendezvous information from the DATA_STATION, while the procedure calls of RECEIVE and PUT_AND WAIT from the called task will also get and send data and other rendezvous information from the DATA_STATION. Figures 4a and 4b shows the difference between the original rendezvous and the emulation rendezvous.

Implementation

According to the concepts described, we can divide the description of the implementation of ARET into three parts: Setting up the DATA_STATION, emulating the entry call, and emulating the accept statement.

Setting up the DATA_STATION: We define the DATA_STATION as a package and set up data structures corresponding to each entry in the DATA_STATION. Since any entry is defined in a task specification, actually this part of the tool preprocesses task specifications in the source code. As soon as the data structures for an entry are set up in the DATA_STATION, the entry definition is eliminated from the source code by adding a comment sign (--) before it.

Figure 4a: A rendezvous achieved by entry call and accept statement

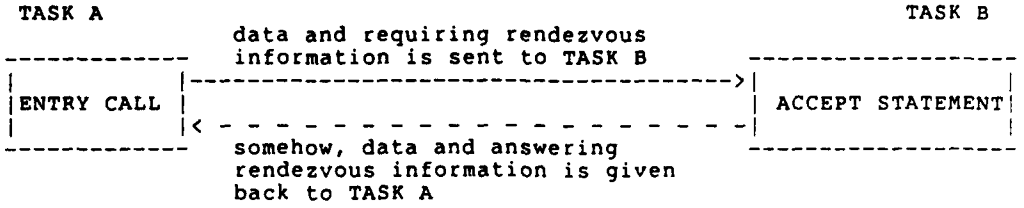
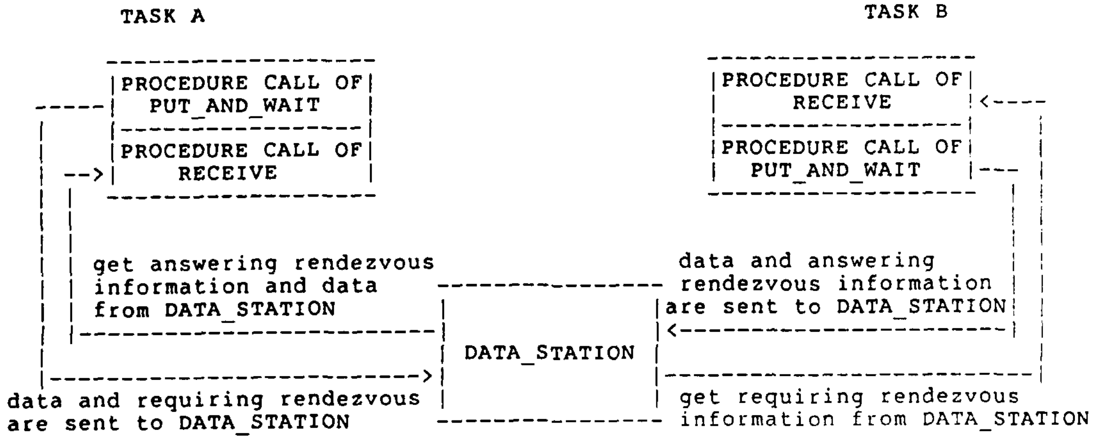

TASK A
TASK B
data and requiring rendezvous information is sent to TASK B
ENTRY CALL
ACCEPT STATEMENT
somehow, data and answering rendezvous information is given back to TASK A

Figure 4b: An emulated rendezvous achieved by procedure calls of PUT_AND_WAIT and RECEIVE.


TASK A
TASK B
PROCEDURE CALL OF PUT_AND_WAIT
PROCEDURE CALL OF RECEIVE
PROCEDURE CALL OF RECEIVE
PROCEDURE CALL OF PUT_AND_WAIT
get answering rendezvous information and data from DATA_STATION
data and answering rendezvous information are sent to DATA_STATION
DATA_STATION
data and requiring rendezvous are sent to DATA_STATION
get requiring rendezvous information from DATA_STATION

The data structures for each entry are set up according to the needs of the procedure calls of PUT_AND_WAIT and RECEIVE, and the types of these data structures are imported from the communications package. The data structures for each entry include:

- A sequence of object declarations: Each object corresponds to one of that entry's form parameters. These objects will be used to contain data transferred from other tasks.

- A sequence of object_logical_addressee declarations: Each object_logical_addressee declaration corresponds to an object which is in turn corresponding to a form parameter of that entry. The reason for setting up this data structure is that the PUT_AND_ WAIT procedure will transfer the data to a physical address through its logical address. The connection of a physical address to its logical address is achieved by the procedure DECLARE LOGICAL ADDRESSEE of the communications package, and this procedure is called in the package body of DATA_STATION. Figure 5 shows how the data structures are devised for an entry.

- An object that represents the amount of data to be transferred by the entry: According to the requirements of the PUT_AND_WAIT procedure, this object should be WORD_COUNT_TYPE which is defined in the communications package. The value of that object is evaluated in DATA_STATION by accumulating the size of all the parameters of that entry. The program segment in Figure 6 shows how that object is devised.

Figure 5

```
entry A (X1: integer;       with COMMUNICATIONS;
         X2: boolean);      use  COMMUNICATIONS;
                            package DATA_STATION is
                                    .
                                    .
                                    .
                              A_X1    : integer;
                              A_X2    : boolean;
                              A_X1_LA: LOGICAL_ADDRESSEE_TYPE:=1;
                              A_X2_LA: LOGICAL_ADDRESSEE_TYPE:=2;
                                    .
                                    .
                                    .
                            end DATA_STATION;

                            package body DATA_STATION is
                                    .
                                    .
                                    .
                              DECLARE_LOGICAL_ADDRESSEE(A_X1_LA,
                                                  A_X1'ADDRESS);
                              DECLARE_LOGICAL_ADDRESSEE(A_X1_LA,
                                                  A_X2'ADDRESS);
                                    .
                                    .
                                    .
```

Figure 6

```
with COMMUNICATIONS;
use  COMMUNICATIONS;
package DATA_STATION is
        .
        .
        .
  WORD_SIZE_A: WORD_SIZE_TYPE;
        .
        .
        .
end DATA_STATION;

package body DATA_STAION is
        .
        .
        .
  WORD_SIZE_A:=0;
  WORD_SIZE_A:=WORD_SIZE_A +
               WORD_COUNT_TYPE(A_X1'SIZE/WORD_SIZE);
  WORD_SIZE_A:=WORD_SIZE_A +
               WORD_OCUNT_TYPE(A_X2'SIZE/WORD_SIZE);
        .
        .
        .
```

Figure 7: DATA_STATION Queues

```
QUEUE 1: monitor the                QUEUE 2: monitor the
         calling tasks                       called task

          ------------                        -----------
          | TASK A1  |- - - - - - - - - - - -| TASK B  |
          |----------|                        |---------|
          | TASK A2  |- - - - - - - - - - - -| TASK B  |
          |----------|                        |---------|
          |    .     |          .             |    .    |
               .                .                  .
               .                .                  .
          |    .     |          .             |    .    |
          |----------|                        |---------|
          | TASK An  |- - - - - - - - - - - -| TASK B  |
          ------------                        -----------
```

—Two logical address arrays: These are used as queues to solve the situation of multiple tasks calling an entry simultaneously. Each task calling the entry will have its first actual parameter's logical address put into the queue used to monitor the calling tasks, while the called entry's first form parameter's logical address will be put into the corresponding position in another queue used to monitor the called task. Figure 7 shows the two queues.

-An object ot logical address type, for reference, we call it ACTIVATED_REASON: ACTIVATED_REASON is used as actual parameter of RECEIVE to get a logical address value which shows the transferred data reaching that logical address. Hence, we can determine whether the rendezvous be successful by ACTIVATED_REASON's value.

Emulating the entry call: We may use a pair of procedure calls of PUT_AND_WAIT and RECEIVE, combined with some loop, case and if statements, to emulate what the entry call should do. The kernel is formed from the procedure calls of PUT_AND_WAIT and RECEIVE. The detailed implementation can be divided into three steps:

1. Set up data structures for each actual parameter of that entry call in DATA_STATION, and form a sequence of assign statements which transfer the values of actual parameters to their corresponding data structures in DATA_STATION. This is done because the procedure call PUT_AND_WAIT can only transfer data in consecutive locations, while the actual parameters may be defined in different places. Therefore, before the procedure call PUT_AND_WAIT, we should copy the actual parameters into consecutive locations in DATA_STATION.

2. Form a procedure call of PUT_AND_WAIT which will transfer the data in the result of (1) to the corresponding data structures of the called entry in the DATA_STATION. As soon as the transfer is finished, this procedure call will set a flag for the logical address on which the data is transferred.

3. Form a procedure call of RECEIVE which is used to wait for a flag given by the called task. When that called task gets the data transferred from the calling task, that flag will be given by a procedure call of PUT_AND_WAIT in the called task which is used to emulate the corresponding accept statement.

Emulating the accept statement: Emulating the accept statement is very similar to emulating the entry call, but there are only two steps.

1. Form a procedure call of RECEIVE which will scan the queue used to monitor the called task. As soon as some data is transferred to the called task, the RECEIVE procedure will capture the logical address from which the data is transferred and put that logical address into its actual parameter ACTIVATED_REASON.

2. Form a procedure call of PUT_AND_WAIT. According to the logical address found in (1) in ACTIVATED_REASON, the PUT_AND_WAIT will send back data to that logical address to inform the calling task that the transferred data is received.

Implementation Problems

The idea of ARET is easy to understand, while the implementation of ARET is far from easy to achieve. Since Ada is one of the most complicated programming languages, it seems impossible to implement ARET without making constraints on the use of the language, a situation that has already attracted negative attention from researchers in this field (see VolzFC). When entry definitions are eliminated from the source code, for example, how can the entry's attributes and representation clause be used? We still face some other problems such as the processing of nested tasks, the processing of renamed tasks and entries, the processing of task objects created by an allocator, the processing of family entries, and so on. The most difficult problem is the type problem, not only in the sense of task types, but also the entry parameter's type.

One of the key jobs of ARET is to create DATA_STATION and set up data structures for entry parameters. If the parameters are of predefined types, things are straightforward. But if the parameters are of user defined types, then we should recognize them and copy them to DATA_STATION. If the parameter's type definitions are derived through multiple levels of user defined type definitions, then we should look back upon each level of user type definition until we reach the base type and copy all the related type definitions to DATA_STATION. If the types of entry parameters are derived from with clauses and use clauses, then the situation gets even much more complicated.

Facing these problems, we have three choices: making ARET more complicated so that ARET can process these problems, giving more constraints on the use of the language, or, the third choice which is a trade_off between choice one and choice two in that some more sophistication is added to the tool to reduce, but not eliminate, the constraints.

Other problems develop due to the simple fact that the tool creates code that emulates, rather than directly accomplishes the rendezvous. Specifically, the size of the resultant source increases, the rendezvous synchronization mechanism does not work as expected, and renaming of entries can not be done.

Increased size of source code: With the tools from the communications package, several procedures are used for each entry and accept statment in the source file to implement the rendezvous. Consider the following for an entry and the corresponding accept statement.

A. For each entry declaration and each parameter in each entry, there will be...

- One formal and one actual declaration for each parameter to hold the physical address.
- Two logical address variables are declared, one for each formal and actual parameter.
- A procedure call, DECLARE_LOGICAL_ADDRESSEE, is set up for each pair of logical and physical addresses that are declared.

In other words, if an entry has n parameters then there would be at least 2 * 2 * n + (2 * n) * (procedure calls).

For each accept statement there will be...

- Two declarations of type LOGICAL_ADDRESSEE_ARRAY_TYPE which hold the logical address of an entry.
- One signal to ACTIVATED_REASON.

For emulating the rendez-vous there will be...

- One declaration for the result of PUT_AND_WAIT.
- The declarations of the total word size of the data that needs to be transfered every time the entry call is made.

For each entry call and accept statement which occur as a rendezvous, then there will be a "mirror" pair of procedures called: PUT_AND_WAIT and RECEIVE and RECEIVE and PUT_AND_WAIT. This will include the lines where the the formal or actual parameters are copied.

Consider the example of emulating a one line entry call in Figure 8:

Figure 8

It is obvious that even a single entry results in a fair increase in source:

```
Original  :  Ping.enter_ping ( signal );              -- one line entry call.

Emulation :

  Entry_enter_ping_al_pa := signal;                   -- copy actual parameter
  loop
    Put_and_Wait ( Entry_enter_ping_fl_la,             -- formal parameter.
                   Entry_enter_ping_al_pa'address,     -- actual parameter.
                   Word_size_enter_ping,               -- total wordsize.
                   1,                                  -- priority.
                   Enter_ping_result );                -- result.
    Case Enter_ping_result is
      when Successful => exit;
      when others     => null;
    end case;
  end loop;
  loop
    receive ( Entry_enter_ping_out_la,
              1.0,
              enter_ping_activated_reason );
    if enter_ping_activated_reason = entry_enter_ping_out_la(1) then
      Signal := entry_enter_ping_al_pa;          -- copy back the value.
      exit;
    else
      null;
    end if;
  end loop;
```

and the declarations will be :

```
Entry_enter_ping_fl_la : LOGICAL_ADDRESSEE_TYPE := 1;
Entry_enter_ping_fl_pa : boolean;
Entry_enter_ping_al_la : LOGICAL_ADDRESSEE_TYPE := 2;
Entry_enter_ping_al_pa : boolean;
Entry_ping_activated_reason : LOGICAL_ADDRESSEE_TYPE;
Entry_ping_out_la : LOGICAL_ADDRESSEE_ARRAY_TYPE(1..1);
Entry_ping_in_la  : LOGICAL_ADDRESSEE_ARRAY_TYPE(1..1);
Word_size_enter_ping : Word_count_type;
Enter_ping_result : Result_type;
DECLARE_LOGICAL_ADDRESSEE ( entry_enter_ping_fl_la,
                            entry_enter_ping_fl_pa'address );
DECLARE_LOGICAL_ADDRESSEE ( entry_enter_ping_al_la,
                            entry_enter_ping_al_pa'address );
Word_size_enter_ping : word_count_type ( entry_enter_ping_fl_pa'size
                                         word_size );
entry_enter_ping_in_la(1) := entry_enter_ping_fl_la;
entry_enter_ping_out_la(1) := entry_enter_ping_al_la;
```

<u>Differences in synchronization:</u> Consider the diagram in Figure 9 which shows the nature of the "passive side" and "active side" services of the communications package used in the rendezvous.

Problems arise in two areas, the beginning and the ending of the rendezvous:

case 1: (they meet)

We can see that in the emulation task A always waits for task B to call it, otherwise task A cannot continue. On the other hand, task B will not stop even though it is ready to start the rendezvous.

case 2: (they leave)

As opposite to when they meet, task A is left before task B is waiting and receives the message to leave after task A is on its way to being active again.

Figure 9

```
Ada rendezvous:

            task body A is                task body B is
                   .                             .
                   .                             .
                   .                             .
              accept AE do                      A.AE;
                   .                             .
                   .                             .
                   .                             .
              end A;                      end B;

          end A;

Emulated rendezvous:

          task A;                        task B;

          RECEIVE                      PUT_AND_WAIT
             .                              .
             .                              .
             .                              .
        PUT_AND_WAIT                     RECEIVE
             .                              .
             .                              .
             .                              .
       end task A;                     end task B;
```

Figure 10

```
               activated        receive                              put
     task A   |- - - - - - - - -|--------------------------------|- - - - - - -|
                                 ^                                 ^
                                 |           rendezvous            |
               activated         |                                 |
     task B   |- - - - - - - - -|- -|------------------------------|- - - |
                                 put receive
```

This follows the timing diagram in Figure 10.

The reason that the rendezvous timing is off is because the put never waits while the receive has to receive the information and copy back the values.

Renaming is unimplementable: According to DoD83 the task "...entries may be overloaded both with each other and with subprograms." However, in ARET, entry names are discarded and logical addresses are introduced instead. Every entry call uses the same two procedures, and the only difference between two entries is the logical addresses that are passed in the parameters. Therefore, rename an entry, there must be an entry name but since entry names are discarded then the feature of renaming is lost. Also because entry names are discarded, overloading entry names is, too, a problem. Procedure calls and entry calls are made in the same structure, therefore the ambiguity will arise in cases such as default value parameters. For instance:

```
procedure AAA ( X : T1;
                Y : T2;
                Z : boolean := false );
entry     AAA ( X : T1; Y : T2 );
```

if the call is made such that

AAA (X1, Y1); where X1, Y1 are of type T1, T2, respectively.

Currently ARET works within the constraints outlined in this section. The continuation of ARET's development is to conduct case studies of the use of the tool in creating embedded real time software in order to compare its efficiency with the "small Ada program" paradigm of task distribution presently in use.

Bibliography

[Armitage85]

James W. Armitage and James V. Chelini, "Ada Software on Distributed Targets: A Survey of Approaches," in ACM Ada Letters, Vol. IV, 4, Jan/Feb 1985, pp. 32-37.

[Booch86]

Grady Booch, Software Engineering with Ada. Menlo Park, CA: Benjamin/Cummings Publishing C., Inc, 1986.

[Buzzard85]

G. D. Buzzard and Trevor N. Mudge, "Object-Based Computing and the Ada Programming Language," in IEEE Computer, March, 1985, pp. 11-19.

[Cornhill83]

Dennis Cornhill, "A Survivable Distributed Computing System for Embedded Application Programs Written in Ada," in ACM Ada Letters, Volume III, 3, Nov/Dec 1983, pp. 79-87.

[DoD83]

Reference Manual for the Ada Programming Language U. S. Department of Defense, January, 1983.

[Eisenhauer86]

Greg Eisenhauer, Rakesh Jha, and J. M. Kamrad, "Distributed Ada: Methodology, Notation, and Tools," in the Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station, June, 1986, pp. B.3.2.1-8.

[Fuller78]

Samuel H. Fuller, et al, "Multi-Microprocessors: An Overview and Working Example," in Proceedings of the IEEE, Vol. 66, 2, Feb 1978, pp. 216-228.

[Gehani84a]

Narain Gehani, Ada: Concurrent Programming. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[Gehani84b]

Narain Gehani and T. A. Cargill, "Concurrent Programming in the Ada Language: the Polling Bias," in Software---Practice and Experience, Vol. 14, 5, May 1984, pp. 413-427.

[Habermann80]

A. Nico Habermann and Issac R. Nassi, "Efficient Implementation of Ada Tasks," Carnegie Mellon University Department of Computer Science Technical Report CMU-CS-80-103.

[Jones78]

Anita K. Jones, et al, "Programming Issues Raised by a Multiprocessor," in Proceedings of the IEEE, Vol. 66, 2, Feb 1978, pp. 229-237.

[Jones80]

Anita K. Jones and Peter Schwartz, "Experience Using Multiprocessor Systems --- A Status Report," in Computing Surveys, Vol. 12, 2, June 1980, pp. 121-165.

[Knight87]

John C. Knight and John I. A. Urquhart, "On the Implementation and Use of Ada on Fault-Tolerant Distributed Systems," in the IEEE Transactions on Software Engineering, 1987.

[McDermid82]

John A. McDermid, "Ada on Multiple Processors," Royal Signals and Radar Establishment, Controller HMSO, London, 1982.

[Mundie86]

David A. Mundie and David A. Fisher, "Parallel Processing in Ada," in IEEE Computer, August 1986, pp. 20-25.

[Organick72]

Elliot I. Organick, The Multics System. Boston: MIT Press, 1972.

[Roberts81]

Eric S. Roberts, et al, "Task Management in Ada --- A Critical Evaluation for Real-Time Multiprocessors," in Software --- Practice and Experience, Vol. 11, 1981, pp. 1019-1051.

[Rogers86]

Patrick Rogers and Charles W. McKay, "Distributing Program Entities in Ada," in the Proceedings of the First International Conference on Ada Programming Language

Applications for the NASA Space Station, June, 1986, B.3.4.1-13.

[Siewiorek78]

Daniel P. Siewiorek, et al, "A Case Study of C.mmp, Cm*, and C.vmp," in the Proceedings of the IEEE, Vol. 66, 10, Oct 1978, pp. 1178-1199, and pp. 1200-1220.

[Swan77a]

Richard J. Swan, et al, "Cm* --- A Modular, Multi-Microprocessor," in the AFIPS Conference Proceedings 1977, Vol. 46, pp. 637-644.

[Swan77b]

Richard J. Swan, et al, "The Implementation of the Cm* Multi-Microprocessor," in the AFIPS Conference Proceedings 1977, Vol. 46, pp. 645-655.

[Tedd84]

Mike Tedd, et al, eds., Ada for Multi-Microprocessors. Cambridge: Cambridge University Press, 1984.

[Volz85]

Richard A. Volz, et al, "Some Problems in Distributing Real-Time Ada Programs Across Machines," in Proceedings of the Ada International Conference, 1985, Cambridge University Press, 1985.

[Volz87]

Richard A. Volz and Trevor N. Mudge, "Timing Issues in the Distributed Execution of Ada Programs," in IEEE Transactions on Computers, Volume C-36, 4, Apr 1987, pp. 449-459.

[VolzFC]

Richard A. Volz, et al, "Translation and Execution of Distributed Ada Programs: Is It Still Ada?" forthcoming in IEEE Transactions on Software Engineering.

[Wellings84]

A. J. Wellings, G. M. Tomlinson, D. Keefe, and I .C. Ward, "Communication between Ada Programs," IEEE Computer Society 1984 Conference on Ada Applications and Environments, IEEE Computer Society Press, 1984, pp. 145-151.

[Wulf81]

William A. Wulf, Roy Levin, and Samuel P. Harbison, HYDRA/C.mmp: An Experimental Computer System. New York: McGraw-Hill, 1981.

Primary Contact:

Dr. James E. Tomayko, Director, Software Engineering Program, Campus Box 83, The Wichita State University, Wichita, KS 67208

Telephone: 316-689-3156; Arpanet: jet@sei.cmu.edu; Bitnet: JETOMAYK@TWSUVM

Dr. James E. Tomayko is an Associate Professor in the Computer Science Department and Director of the Software Engineering Program at The Wichita State University. During 1986-1987 he was a Senior Computer Scientist at the Software Engineering Institute. He has done contract work for NASA and the Boeing Military Airplane Company, and provided consulting for numerous organizations. Dr. Tomayko is a Distinguished National Lecturer for the Association for Computing Machinery.

Wei Wang is a Visiting Scholar at The Wichita State University. Mr. Wang's home institution is the Beijing Polytechnical University in China.

Minh Leo Pham is a graduate student in The Wichita State University Department of Computer Science. He received his B. S. in Computer Science from WSU in 1987.

Jim Kroening is a Senior Software Engineer at Boeing Military Airplane Company in Wichita, KS. He has designed and developed operating systems for real time avionics applications for seven years. He will complete a master's degree in Computer Science at Wichita State in May, 1988.

DISTRIBUTED ADA: EXTENDING THE RUNTIME ENVIRONMENT FOR THE SPACE STATION PROGRAM

Charlie Randall
GHG Corporation
Houston, TX 77058

Patrick Rogers
Charles W. McKay
Software Engineering Research Center
University of Houston - Clear Lake
Houston, TX 77058

Abstract

NASA's Space Station will be the largest distributed system ever undertaken. For the project NASA has selected Ada as its programming language. We are investigating the use of Ada for distributed programming as it will be required for the Space Station. As a framework for this work we are using the Clear Lake Model for Runtime Support Environments.

1. Introduction

Distributed processing is a major concern for the Space Station Program (SSP) because of its multitude of ground and space based, interconnected systems. The NASA Johnson Space Center (JSC) is investigating the Clear Lake Model for Ada Runtime Environments as a framework for the SSP. *The University of Houston -Clear Lake (UH-CL) High* Technologies Lab and GHG Corporation are researching the model and developing a proof-of-concept prototype on the JSC Distributed Ada Testbed.

This project is based on findings of the Joint NASA/JSC UH-CL APSE Beta Test Site team while studying the issues of applying and evolving the toolset for the Minimal Ada Programming Support Environment (MAPSE) to large, complex, nonstop, distributed applications like the SSP.

2. Background

Recognizing the lack of definition of an Ada Runtime Support Environment (RTSE), the Beta test site team developed a conceptual model, the Clear Lake Model for Ada Runtime Support Environments, defining an extensible Ada RTSE that can be used for single processor, multiprocessor, and networking targets. The model proposes an interface to a virtual Ada machine for the object code of the applications software and the command language.

2.1 Runtime Support Environment

In the Clear Lake Model, between the interface to the virtual Ada machine, the processor, and the associated hardware resources, are the following three types of RTSE software resources: the runtime kernel (RTK), the runtime library (RTL), and the extended runtime library (XRTL).

The RTK masks the idiosyncrasies, i.e., the machine dependencies, of the particular processors from the RTL and XRTL. The RTK is also responsible for sequential management activities that permit the processor to service the application code via the RTSE. Unlike the components of the RTL and XRTL which are designed and developed in Ada, execution speed requirements and hardware dependencies for the RTK will require unique, processor-dependent optimizations of each RTK.

The RTL is responsible for the minimal set of runtime support required by the Ada Language Reference Manual, ANSI Mil-Std-1815A. In contrast, the XRTL routines provide functionality beyond that defined in the Reference Manual. It consists of runtime support modules specifically tailored to the requirements of the applications and systems software. For example, a particular scheduler may be needed for one subsystem, while another subsystem may require multi-level security. Some of these subsystems may need performance monitors and diagnostic aids. To provide selectability and configurability, some library functions will have multiple implementations. Some of these functions are for utilization on different processors. Others simply have diverse operational requirements and constraints In spite of *these different requirements*, most of these RTSE services should be transparent to the applications code.

2.2 Clear Lake Model

The Clear Lake Model for RTSEs addresses the distribution of any Ada entity (tasks, variables, procedures, etc.) over geographically dispersed systems [Randall]. Using the XRTL approach, the language is "extended", as prescribed in the Language Reference Manual, by creating services and resources to provide the necessary additional functionality, specifically for distributed processing.

The context of execution for distributed Ada programs will include execution on machines that are geographically dispersed, such that communication can not be via shared memory or a shared bus. This situation implies the need for a model that supports distribution of entities over the entire distribution spectrum in as uniform a manner as possible, based on cooperating runtime systems that communicate by messages. In the Clear Lake Model, this support is provided by "surrogates". Surrogates are independent "processes" within each runtime system that reside on each of the distributed processors in the target environment. They manage the *cooperation required by the distributed programs*. Activities that require remote access go through these surrogates in a manner transparent to the applications program. These remote accesses range from references to individual program variables that are located on another machine, to library unit references that are likewise remotely located. The surrogate-based approach handles the entire range, in a consistent manner.

For instance, a reference to a remotely-located variable requires message to be sent to the RTSE of that remote machine. That RTSE has its own surrogate process, which receives and "parses" the incoming message. Once parsing is complete, the local surrogate process dedicates a local agent to perform the action indicated in the message, for example "get the current value of variable X". This local agent is scheduled for execution like any local task, and when dispatched, performs whatever steps are necessary to fulfill the request. Such activities include determining the location of X (e.g., what package it is in), getting the current value, and sending a message to the originating RTSE containing that value. Once the agent has finished, it is returned to a pool for further use at a later time.

At the other end of the distribution spectrum, the surrogate approach still applies (thus the consistency mentioned earlier). For example, a call to a remote subprogram can be supported in the same manner as a remote variable reference: a message is sent to the remote RTSE, the surrogate manager dedicates a local agent to make the subprogram call and, upon return from the call, a message to the originating RTSE is returned to allow the caller to continue. Buffering of parameters would be done in a manner commensurate with the semantics specified by the Reference Manual. This would require more messages, but would not require additional functionality orthogonal to the surrogate-based approach. Note that the activities of a "local" agent may in turn generate further messages to "remote" sites, in order to execute the current request. This may continue indefinitely, until all references are satisfied.

A surrogate is implemented as a task of a particular Ada language task type on each of the distributed processors that manage the remote references. A network is assumed for communications since the machines involved may be geographically separate (e.g., in orbit and on the ground). The pseudocode for the surrogate task type follows:

```
begin
   loop
      select
         accept a request to send a
               message to a remote site
               (this may come from an
               application program or a local
               agent allocated to service a
               remote request)
      or
         accept an incoming message from a
               remote site
         dedicate a local agent to service
               the request
      end select
   end loop
end Surrogate
```

3. Distributed Systems and Processing

There are several issues involved when considering distributed systems. Some of these are unique to distributed systems. But others are derived from network systems, which are a restricted form of distributed systems. It is assumed that the SSP will have an underlying network system connecting the many different interacting subsystems.

The following items have been noted as three dimensions that parameterize distributed system [Volz]: memory/processor organization, binding time (i.e., whether it occurs at compile time, between the front end and back end of compilation, at link time, or at runtime), and the degree of homogeneity/heterogeneity of the processors in the system. Consideration of these along with the needs of the SSP leads to the following conclusions. Due to its many and highly dispersed subsystems, the idea of shared memory is impractical for the SSP and thus it will be a loosely coupled system. We envision using DIANA, the "standard" intermediate language for Ada, as the principle mechanism for implementing the distribution scheme. Therefore most of binding time will occur between the front end and the back end of the compilation when the DIANA becomes available. Regarding the last of these dimensions, the SSP consists of many different subsystems and will have a high degree of processor heterogeneity. During its existence of thirty years, and probably longer, the SSP will need to add new and/or different hardware technology to accommodate evolving needs. This is one of the main assumptions upon which the Clear Lake Model is based.

In addition to the above three items, there are several other matters relevant to any discussion of distributed systems [McKay]. The first of these is the perception of time. The issue of time involves more than the minimal set of guarantees that Ada provides for time or that Ada has no requirements on upper bounds for time. The main concern here is that it is not possible to guarantee that all of the spatially dispersed points within the system will be synchronized. Time will have to be approached from totally different control perspective. It is assumed here that the underlying network system will take care of most of the time synchronization problem, but still the designers of any affected application programs will have to be aware of the problem and the constraints it may cause.

Another issue is the reliability of communications. Again the network is assumed to provide adequate communications and related status information. The designer in this case can then proceed knowing that either any required communications will be carried out or that notification of its failure or inability to complete will be indicated. Although the ISO/OSI model will sufficiently handle most of these network communication services, it is recommended that the Ada rendezvous be added for the purpose of fault tolerance.

In considering what the network will provide, we have tried not to make over-simplifying assumptions. The assumptions are driven by what we feel are reasonable beliefs about what network can and should provide. At the same time we have tried not to circumvent any of the hard issues by the use of judicious, unrealistic assumptions. Such an example is the notion of a fail-stop processor. A processor does not fail/stop nicely in a critical situation. It fails in a flaky way. It does not surrender buses voluntarily or act in any other desirable manner. The design therefore must encapsulate any such critical situation. It must provide firewalling in such a situation so that the processor involved can be forced off the bus if that is the desired affect. We consider the above network systems assumptions to be consistent with those of the SSP.

Safety and security are another important aspect of any distributed system (or for that matter any system upon which life and property depend). The system must guard itself against any action, intentional or accidental, that attempts to compromise its integrity. This includes considering safety and security requirements at each point of the system's lifetime. A safe system is able to monitor its situation and detect faults that enter the system state vectors as soon as possible, firewall their propagation, analyze their effects, and recover safely. The use of Ada and a well designed RTSE provide exceptional assistance towards this goal.

An additional item concerning Ada and distributed systems is that Ada does not need to be modified for use with distributed systems. It is not that there are some enhancements or improvements that we would like to have. But we have not found anything fundamentally wrong with using the current version of Ada for distributed systems. Basically we feel that the RTSE concerns are more important than the language issues.

One assumption we do make is that the current version of the Ada language need not be modified for use in distributed systems. It is not that there is no room for improvement for the language in this area, but we have not found anything fundamentally wrong or too cumbersome with the use of Ada for distributed programming.

4. Methods of Distribution

For any system there are services and resources provided for use within the system. These constitute two different aspects of the same thing, i.e., the available facilities. Services are the active element (e.g., timing functions) and resources the passive element (e.g., memory). If a set of these services and resources are to be shared by more than one application program then access to them should be provided through the system software. If on the other hand, only one application program needs a set of services and resources, then that application should provide access to those services and resources. Shared resources and services then are provided via the system, unique resources and services are handled within the application making use of them.

4.1 Distribution Visibility

Under the Clear Lake Model, distributed processing is provided through a set of services and resources. Again, that part of the distribution that is unique to the application should be provided by the application while that part of the distribution that is sharable should be provided by the system, in this case the runtime support environment.

There have been two basic approaches to the interface between application programs and the system in respect to distribution. We call this distribution visibility based on whether a program knows internally the distribution scheme. The first approach, taken by Honeywell [Cornhill] [Kamrad], assumes that the program is unaware of any distribution, especially in light of fault tolerant reconfiguration. This is transparent distribution because the system handles all aspects of the distribution, reconfiguration, etc. and the program is not aware of any of the distribution. The Honeywell approach does allow for some limited control (non-functional) over some aspects of the distribution through their specification written in the Ada Program Partitioning Language (APPL). The advantages are that the software can be designed and implemented without regard to distribution, thus providing greater system flexibility. The functional requirements of the program are handled separately from the non-functional ones. One benefit of this is obvious when the software is later moved to another system. Because the design was done in separate stages, the functional requirements design portion of the development should be minimally impacted and most changes should affect only the distributed specification, which accommodate most of the hardware specific design decisions. The disadvantages include increased overhead where ever distribution is a possibility and no way (or at best a very limited way) to specify how a program can provide safe or degraded service following a processor failure.

In the second approach, the programmer specifies everything concerning distribution within the program [Knight]. Thus the visible (or programmer-controlled or non-transparent) approach makes distribution totally the responsibility of the programmer. Within the program, there is code to address all of the distribution concerns, especially those regarding reconfiguration. The advantage of this approach is the minimal overhead, and then only where it is necessary. Also services provided after a failure need not be the same as those provided had no failure occurred. Thus degraded service can be accommodated. The disadvantages are that the programmer must provide for all the possible distributions or assume the system can handle whatever is not specified. Also, the code generated will probably be machine specific, thus possibly limiting its use.

The Clear Lake Model strives to combine the merits of both approaches. Those services and resources required of the distribution scheme that are not unique to any application program are handled by the system, i.e., the RTSE. This is the situation for which the Honeywell approach is appropriate. And those distribution services and resources that are specific to any one application program, e.g., providing its own unique degraded service, are handled from within that application program. This is the situation for which Knight's approach is appropriate. If a programmer is concerned about how to distribute the program, what response to make to certain failure, or that the default distribution and its associated overhead do not meet requirements, then providing for distribution becomes part of the program implementation. Otherwise the program is implemented without regard to distribution.

The difference between Knight's approach and that of the Clear Lake Model is that Knight essentially uses dynamically allocated Ada tasks as the means of expressing responses to failures. By "dynamically allocated", we mean that the predefined constructs for dynamic allocation (i.e., "new type_mark") are used. In contrast, the Clear Lake Model makes direct use of the underlying runtime system (RTS) to interrogate the state of individual program components and the system as a whole, as well as to direct the specific response. The access to the RTS is provided by Ada packages that are imported from the XRTL. These RTS supported fault tolerance capabilities are required, regardless of the approach advocated, since the application program cannot be expected to provide them. The Clear Lake Model merely makes them available to application programs.

Additional, Knight's approach does not address those situations when the user specified distribution can not be allowed. In some situations, when the application programmer has specified the distribution scheme, system requirements and responsibilities may overrule those specifications. When issues such as safety and security are involved, the system may have to cancel, or limit, the programmer's control over distribution.

Honeywell's partitioning is described by a specification written in APPL. The APPL specification indicates which portions of the program should be partitioned onto which distributed processors. Honeywell feels that the APPL approach is better than specifying partitioning through the

PRAGMA construct for two reasons. First, the PRAGMAs are embedded in the source and thus must be scattered through out the program. This makes library unit sharing very inconvenient. The APPL specification would be centrally located, making it easier to find and form a global picture of the system. Secondly, there is some concern over whether PRAGMAs are extensible to the specification of non-functional requirements, i.e., for reconfiguration and fault tolerance.

While we agree that PRAGMAs are not the way to approach the problem of specifying distribution, we do not use APPL. We feel that APPL does not allow a look outside the software world. Instead something else is needed to represent such things as the non-functional requirements involving hardware and operating systems in a manner consistent with the EA/RA (entity attribute/relationship attribute) model. We purpose the use of DIANA and extending it by adding attributes representing non-functional requirements. The non-functional requirements would be given in a separate specification and/or through the interaction with specification tools.

After the program has been completely tested in the host environment it is ready to begin the distribution process. Its DIANA representation is used in this phase. DIANA is used because its EA/RA structure allows it to be easily modified by tools for distribution. Specifically this is done by adding attributes, in this case ones representing the non-functional requirements, to the DIANA tree. The first of the tools is the Partitioning and Allocation Tool. Taking the non-functional information, it maps each of the entities that the programmer has marked as distributable to its symbolic location, adding this information also to the DIANA tree. When it is time to build the actual distributed system, another tool takes the information in the DIANA tree, notes the target machine resources such as the the machine instruction set architectures, pulls in the necessary routines from the runtime libraries and builds the load modules for each of the processors in the particular distribution scheme.

4.2 Level of Distribution

Having established distribution visibility, the next question is at what level does distribution occur? That is, is it best to have a separate program per processor or a single program distributed over the different processors. The separate program approach represents the old way of doing things. Although this approach is the easier, at least in the short term, with the advent of the Ada programming language it becomes highly undesirable. Use of a single program, specifically one written in Ada, allows for a greater assurance of boundary/interface consistency through compile time checks, strong typing of variables, communications between processors is hidden resulting in a less complex program, and a greater control over parallel execution is provided. For the SSP both levels will have to be used.

5. Unit of Distribution

If it is best to distribute parts of an Ada program between processors, then the obvious question becomes which parts to distribute. There are trade-offs to be considered with any unit of distribution. In some cases the overhead is too great and/or there is no possible benefit from the distribution, e.g., distributing individual statements. In general the smaller the unit of distribution, the greater the overhead required. The amount of overhead must not be so high as to negatively impact the performance of the software. On the other hand, the larger the unit of distribution, the smaller the opportunity to fully exploit distributed processing which, for example, affects performance. The question then becomes whether the overhead is worth the benefit gained. The answer depends upon the application, only the designer/programmer is in position to judge.

The obvious unit choice is the task. The Language Reference Manual [DOD] makes references to distributing tasks to different processors. Using this approach, anything that needs to be distributed is encapsulated within a task. Individual tasks are then distributed to separate processors through commands via pragmas or representation specifications [Cornhill]. Others [Knight] have suggested using tasks but that Ada does not provide enough support for their distribution. One problem with this approach is the forced encapsulation of the desired distribution element within a task even when a task does not meet the true semantics of the situation.

Another choice is that of library subprograms and library packages [Mudge]. At this level, the only visible entities are library unit identifiers, subprogram parameters for library unit subprograms, and library unit packages' visible declarations. This offers a reasonable, but limited, level of granularity without too much overhead which is relatively easy to distribute. In addition, if tasks need to be distributed, then they can be encapsulated within packages with very little semantic loss.

We have chosen to go beyond these, choosing any reasonable Ada entity as the unit of distribution [Rogers]. Although there can be great overhead in allowing such a fine resolution of distribution, the programmer has control over what is to be distributed and thus can decide whether the overhead incurred is worth the gains made through distribution.

While this choice requires a great deal of work from the XRTL, we feel that if we can provide the user with this fine a control over distribution, then he can easily scale-down to the level that the others above have suggested for the unit of distribution. This is the so called scaling direction problem [McKay]. Too many times in the past, researchers and developers have taken the easy or most general approach to a problem. Later when they attempted to move to the more difficult issues and tried to scale up their previous solutions, they found it difficult or impossible to proceed with their initial assumptions and work. By tackling the difficult or most all encompassing problems first, then it is easy to scale down to these easy or general approaches. A subset of the effort necessary to distribute a single variable should take care of distributing a package or task.

6. Research

Our first step in the development of a prototype to demonstrate the Clear Lake Model was to develop a program to study the steps required for interprocessor communications and fundamental distributed programming. The program is a variation of the bouncing ball program (a ball on the display screen travels in a straight line until it encounters one of the screen's sides; there it either bounces off or is passed to another screen). The program runs between different and physically separate machines and thus different processors. The bouncing ball idea was chosen for its graphical appeal. It easily conveys the notion of communications between the processors by following the ball. For the project we are using the JSC Distributed Ada Testbed, which consists of one Data

General MV/8000 and three Data General MV/2000's. The machines are interconnected via an Ethernet connection using Data General's Internet (TCP/IP) network software.

The identical program executes for this demonstration on each of the machines in the system. A front end to the program allows the user to identify on which machine the program is executing. The user is also asked to identify which of the display terminals to associate with the machine in question. Each machine has one terminal. Additional those terminals are given an placement order, i.e., which one is on the left end, right end, or in the middle. This provides the program with the information it needs when the ball encounters one of the terminal sides. Appropriately, if the ball is at the left side of the left end terminal or the right side of the right end terminal, the ball bounces back from that side. Otherwise the ball is passed to the next machine. More accurately, information representing the ball is passed to the correct machine.

The program was built in several layers. The lowest layer provides the interface to the communications and network software. The next layer over this one is the important one for this research. It serves as an interface between the normal functions of the application code and the underlying distribution scheme. It is the equivalent of the virtual Ada machine for this demonstration. Calls are made to the routines in this layer for any services or resources associated with distribution. Under the scheme planned for the prototype, these routines would be the in the XRTL. At a high level, each of these routines does the same thing. It takes a request and determines which processor in the system can best handle the service or request.

Several consequences of this experience should be noted. We were able to prove that it is relatively trivial to create routines which can call other routines on the same processor or different processor depending upon the prevailing distribution conditions if they are designed and implemented with some forethought. Using layers, instead of levels (as in Unix), leads to a more consistent and safe system. By creating a hierarchy of interfaces that enforces access via the interfaces, individual applications are prevented from using the system services and resources in a current-machine-and-configuration dependent way. The result is a system of greater integrity that is easier to test, enhance, and extend. Our first attempt at the demonstration program only had left end and right end terminals, but it was very easy to add the middle terminals to the system.

The next major step in this research is to build a full prototype to demonstrate the appropriateness of the Clear Lake Model. Our first attempt at this will be to take the program demo discussed above and modify it for use with the RTSE. This will require moving some of the code from the program into a prototype RTSE. The lower most level encapsulating the communications and network services will become part of the XRTL. In addition most of the functions of the next layer will also move to the XRTL. Those that are unique to the application will stay in what will become the lowest layer for the application code.

We are currently in the stage of identifying those routines that form the virtual Ada machine interface with regards to distribution and should be included in the XRTL. A catalog of such routines along with formal rules for creating, installing, and sustaining them; any constraints associated with their utilization; and the possible overhead/trade-offs on a typical system will be the result of this work.

We have begun work on the testbed with the Data General system by looking into their system software and their general approach to software development. The prototype will build upon the software they already have in place. One of our goals is to minimize any altering of the existing system software. Instead we envision augmenting the software by the addition of XRTL routines and RTSE tools specific to distribution.

In addition to developing the prototype we are also investigating the means and requirements for adding performance monitoring/analysis to the system. By making use of distributed resources, much of this monitoring/analysis can be supported solely through software without impacting the normal performance of an application. The Clear Lake Model envisions adding these in a manner similar to providing the distribution scheme, via calls to routines within the XRTL. Under test situations, these routines would record the appropriate data for use in monitoring/analysis. The routine may also determine that to meet the extra load, extra processing power is necessary and take the appropriate actions. Under non-test conditions the same routine would be utilized but the routine would recognize the non-test situation and none of the extra activities would occur. Thus to the user, the same interface, i.e., calling the same routine, applies in both situations.

7. Conclusions

This paper has presented the objectives of our research into having the Clear Lake Model serve as the framework for distributed programming for the SSP, the design decisions involved in developing the Clear Lake Model proof-of-concept prototype, and our experiences with the prototype implementation. Although there remain many questions about the correct way to handle distributed programming, especially in relation to Ada, we believe that using the RTSE as prescribed by the Clear Lake Model is the best current means for the SSP.

REFERENCES

Cornhill, D. "Four Approaches to Partitioning Ada Programs for Execution on Distributed Targets", Proceedings of the IEEE Computer Science Conference on Ada Applications and Environments, 1984, pp. 153-162.

DOD, Reference Manual for the Ada Programming Language, 1983.

Kamrad, M., R. Jha, G. Eisenhauer, and D. Cornhill, "Distributed Ada", Ada Letters, Vol. 7, No. 6, pp. 113-115, 1987.

Knight, J. and J. Urquhart, "On the Implementation and Use of Ada on Fault-Tolerant Distributed Systems", IEEE Transactions on Software Engineering, Vol. SE-13, No. 5, May 1987.

McKay, C., D. Auty, and K. Rogers, "A Study of System Interface Sets (SIS) For the Host, Target, and Integration Environments of the Space Station Program (SSP)", SERC (UH-CL) Report SE.10, NCC9-16, 1987.

Mudge, T., "Units of Distribution for Distributed Ada", Ada Letters, Vol. 7, No. 6, pp. 64-66, 1987.

Randall, C., and P. Rogers, "The Clear Lake Model for Distributed Systems in Ada", SERC (UH-CL) SE.9, NCC9-16, 1987.

Rogers, P. and C. McKay. "Distributed Program Entities in Ada", First International Conference on Ada Programming Language Applications for the NASA Space Station, 1986, pp. B.3.4.1 - B.3.4.13.

Volz, R., "Distributed Ada Execution: A Definitional Void", Ada Letters, Vol. 7, No. 6, pp. 70-72, 1987.

Biographical Sketch

Charlie Randall is a software engineer with GHG Corporation. His research interests include distributed programming, software engineering, reusable software, and Ada. He is currently vice-chair of the Clear Lake Area SIGAda group. He received his B.S. in Physics from Louisiana State University in 1980, a M.S. in Physical Science from the University of Houston-Clear Lake in 1983, and is currently working on a M.S. in Computer Science at the University of Houston-Clear Lake.

Patrick (Pat) Rogers is Research Supervisor at the Software Engineering Research Center and a Principle Member of the Ada Runtime Environment Working Group (ARTEWG). He holds a B.S. and M.S. in Computer Science from the University of Houston at Clear Lake. His current research interests include distributed systems, software engineering, realtime systems and Ada.

Dr. Charles McKay is Director of the Software Engineering Research Center and the High Technologies Lab at the University of Houston-Clear Lake where he is also a full professor of Computer Science. He has over 20 years of consulting, research, and development experience in industry, government, and academia. He is the author of three textbooks and numerous papers and reports. He is currently the chairman of the Interfaces Subgroup of ARTEWG.

Beyond Ada® Validation

R. J. Bavier
AT&T Information Systems

L. Murray
Concurrent Computer Corp.

Monmouth College Software Engineering

ABSTRACT

Currently, there are many Ada compilers available that have passed versions of the Ada Validation Suite, however, we have found that this does not guarantee that the code generated by these compilers meets the needs of real-time applications. There are some implementations of the Ada Language that have passed validation but are not useful for implementing embedded real-time applications or large real-time systems. This paper discusses three parameters beyond Ada validation considerations that we feel are important for a "usable" Ada compiler. We define "usable" as providing the ability to implement a class of problems for which Ada was intended, real-time applications.

1. Introduction

Currently, all Ada compilers must pass the Ada Validation Suite to be eligible for use on DOD contracts. The test programs contained in the Validation Suite do ensure that all the features of the language are functional, but they do not test the usability of these features. There are some implementations of the Ada Language that have passed validation but are not useful for implementing real-time embedded applications or large real-time systems.

For a design project in the Software Engineering curriculum at Monmouth College, we designed a 3-D Radar Tracker using Ada as the design language. This radar tracker had stringent timing and performance requirements and was targeted for a multiprocessor shared memory hardware configuration. Our design made extensive use of tasking, using a separate task to track each object in the radars' coverage. This is similar to the model given in the Ada Rationale[1] for the use of packages and tasks. In investigating the functionality of Ada compilers available to us, we found some problems which would prevent our application from being implemented as designed, or from running in "real-time", as designed. This paper discusses three of the problems we encountered with the Ada language and the Ada run-time environment implementation, and their impact on the use of Ada for the implementation of embedded real-time systems.

® Ada is a registered trademark of the U.S. Government (AJPO)

Three areas that are beyond the scope of Ada validation but which we feel are critical for real-time applications are task scheduling, support of shared memory, and executable program sizes. This paper describes how the implementation of these aspects may prevent a truly usable Ada Language implementation for embedded real-time applications. We define a "usable" compiler as one providing the ability to implement one class of problems for which Ada was intended, real-time applications.

2. Tasking

One of the features of the Ada language is support of concurrency through tasking. This gives the programmer the ability to specify that several threads of the program can proceed at the same time. The rendezvous provides a method for synchronizing independent or cooperating tasks and for transferring data between them. Tasking is a backbone of the Ada language.

Many compilers do not implement "true-tasking" in that each Ada task is not a separate task at the system control level. This can prevent true parallel execution and the use of multiple processors. We found that our validated compiler had implemented tasking via coroutines. In this environment a task must explicitly yield control of the system to allow another task to proceed. This also prevents pre-emption by a process with a higher priority. A new "thread of control" is not really created since the caller is suspended until the callee returns control. While the syntax of the program denotes concurrent processing, the implementation is sequential. Designing a real-time system that will be implemented using this compiler would require that each task poll the system to check to see if a process with a higher priority was waiting to execute, or that each task perform its own quantum checking. This puts a heavy load on tasks.

The coroutine approach is even more restricting when the application is targeted for a multiprocessor system. In this environment there is the opportunity for true concurrency, but the effect is that the application runs as a single process on a single processor regardless of the additional available processors.

Many embedded real-time applications require that "true tasking" be supported at the system level, such that each Ada task is a separate OS process and therefore schedulable on multiple processors. The compiler could provide the interface to the system calls for creating and controlling each task rather than providing its own scheduler. This would enable tasks to be separate identifiable and dispatchable units to the system control software, and therefore allow the application to make use of a multiprocessor hardware configuration. This does however require that any system upon which a real-time Ada system is based support creation, scheduling, and communications between tasks at an operating system level. There would also have to be support for communicating between tasks that are running on different processors.

Alternatively, the task scheduling provided by the Ada run-time environment should support at least pre-emptive priority scheduling in order to support the demands of embedded real-time applications.

There is also a need for the ability of tasks created by separate programs to communicate with each other. This allows greater flexibility in the design of systems and is closely related to the shared memory issue.

3. Shared Memory

The main purpose for shared memory in Ada seems to be a replacement for global variables in a single program, with synchronization of access to these variables for multiple tasks. Shared memory support should also provide the ability for sharing memory between separate programs or applications, since many embedded real-time control systems require two or more programs to concurrently access the same data. Shared memory between separately compiled Ada programs is not addressed by the language standard at all. We see this as a major deficiency in the Ada language.

Currently, Ada requires a system to be built as a single large program. There are many applications in the real-time arena where the programmer must design a program to communicate with and to read data from an existing system, without rewriting, recompiling, or even relinking the existing system. The idea of shared memory should be expanded to include the sharing of memory between tasks even if they are started from different programs. Shared memory should also be implemented at the system level with the compiler providing the interface to the system calls.

Shared memory support should also provide access and synchronization mechanisms that would allow some tasks to have write privileges and some tasks to have read privileges. This mechanism would have to perform arbitration and synchronization so that a task has exclusive access to the shared memory. That would allow a segment of shared memory to be updated with exclusive access and relieve the programmer of the need to write their own semaphores to provide this service.

4. Program Size

The third concern we had was the size of the executable that is generated by the compiler. Program size is very important to embedded systems, since memory size restrictions often require that the executable code be as small as possible. The size of an executable Ada program image may be needlessly large because of the inclusion of unreferenced library routines, runtime support modules, or large default stack and heap sizes.

Real-time systems consisting of many tasks also require reasonably-sized executables. If tasks are very large, the operating system may spend a considerable amount of time swapping tasks in and out of memory. This CPU time would be better spent executing the application.

The tests we conducted on our validated Ada compiler revealed a minimum task size of 1 megabyte. This is unacceptable for embedded applications with strict memory requirements as well as for many multi-tasking real-time applications executing on shared memory multi-processor hardware configurations. Although our machine had 16 megabytes of memory, because of the large task sizes, an application using more than 15 tasks required support for paging or virtual memory in the underlying operating system.

A good compiler and linker should be able to produce an executable that contains only those library functions that are referenced. Any routine or object that is not referenced by the program whether direct or indirect should not be included in the final result. The main thrust of optimizing compilers to generate compact efficient code can be totally wasted when a program of small size requires a large amount of memory for routines that will never be called.

Stack space and additional data areas should also be allocated minimally. Usually, producing a minimal image size results in longer link time. This tradeoff between link time and image size should be available to the user. An Ada environment should at least give the user a link or load time option to increase or decrease the default stack size and allow building minimal size tasks when memory limits are critical.

5. Conclusion

Clearly, more than the current Ada validation procedures are needed to insure a compiler provides a usable implementation of Ada for embedded real-time applications. The run-time environment must support efficient concurrent processing without large memory requirements. The Ada Compiler Evaluation Capability (ACEC) is a step in this direction but the suite should be expanded to thoroughly test the implementation of tasking.

In addition, more thought should be given to language support of shared memory. Ada is missing a standard interface for this valuable feature.

REFERENCES

1. Rationale for the Design of the Ada Programming Language, SigPlan Notices, June 1979.
2. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, U.S. Department of Defense, 1983.

Biographies of the Authors

Richard J. Bavier
AT&T Information Systems
307 Middletown-Lincroft Rd.
Lincroft, N.J. 07738

graduated from Monmouth College in 1985 with a BS in Computer Science. He received his MS in Software Engineering from Monmouth College in 1987. He has worked for AT&T since 1980 starting in Bell Laboratories and transferred to American Bell - AT&T Information Systems. At AT&T he has been involved in development support, system architecture, and software development.

Laurie Murray
Concurrent Computer Corp.
106 Apple St.
Tinton Falls, N.J. 07724
MS 313

is a Senior Member of the Technical Staff at Concurrent Computer Corporation in Tinton Falls, New Jersey. She has been with Concurrent since 1981, and is currently working in the Operating Systems Development group on the company's new proprietary real-time operating system. Laurie has a BS in Computer Science and recently received her Masters in Software Engineering from Monmouth College, West Long Branch, New Jersey.

An Evaluation of Ada Oriented Design Languages [1]

Frances Hunt
Prabhaker Mateti
Amitava Datta[2]
Kouakou Diby[2]
Fuyau Lin [2]

Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio 44106

Abstract

In this report, we evaluate a few languages claiming to support design activity ultimately leading to implementations in Ada. Such languages are known as Ada oriented Design Language (ADLs). The following languages are included in this evaluation: ANNA, Ada/SDP, PDL/81, PDL/Ada, Arcturus PDL, and Byron.

Our evaluation procedure started with a careful study of all available information on each language in the above list. The next step was to rewrite two common examples of "design" in each of the languages. As a result, we (often) revised our opinions about the languages.

When the evaluation process was complete for all the languages we chose, we rated each language based on a number of different criteria. These criteria included many areas, such as: design expressibility, design mapping, rapid prototyping, formal syntax and semantics, understandability, and how well the language's constructs related to Ada. The complete set of criteria and ratings is included in tabular form in the report.

1 Introduction

This report is a survey of a few languages claiming to support design activity ultimately leading to implementations in Ada. Such languages are known as Ada oriented Design Languages (ADLs).

Our evaluation procedure was as follows. We would carefully study all available information on a given language. We would then ask ourselves such questions as: Is this a design language? What aspects of design does it express well? Is it Ada-oriented? Keeping preliminary answers to these in mind, we rewrote two running examples of "design" in each of the languages. As a result, we (often) revised our opinions about the languages.

Section 2 describes what one can realistically expect of present day ADLs. It is by these criteria that we evaluated the ADLs. Section 3 presents a capsule description of each of the languages. Section 4 contains a table of comparisons along with our explanation of what the ratings mean and what the criteria were.

2 What can be expected of an Ada Design Language

In this section, we list expectations that are realistic and feasible today, without requiring major breakthroughs. There are two other such compilations of requirements of ADLs: IEEE Std [9], and the NAC [5]. These two reports cover mostly documentation oriented issues and seem to be driven by the DoD-STD-2167 [7].

2.1 Design Basis

All designs rest on a body of knowledge. Software models the objects of the real world in a symbolic form. A precise description of the models of objects, and perhaps an informal description of how the model relates to the real world object should appear under this heading. We also expect to see a list of their properties. A design language should not only permit but should be so natural that it encourages the expression of these "principles of operation."

In general, this necessitates the importation into the design language of new notations appropriate to the domain of the software.

2.2 Architecture

All designs are compositions of parts. Architecture deals with the elegance and technical properties of different compositions as well as the nature of the components. All designs, regardless of the methodology followed in arriving at them, can be described, perhaps a posteriori, as a series of (i) details, (ii) alternatives considered, and (iii) the reasons for the final choice made.

2.3 Detailed Design

The detailed design of an object is only a few steps away from a concrete implementation of the object. A competent software engineer should be able to "translate" a detailed design into a given programming language almost mechanically, and following some guidelines.

A detailed design of software has the appearance of a program but with (very) high level data types and control structures. These types and control structures are such that standard, although inefficient, ways of implementing them are known.

2.4 Machine Processibility

We take it for granted that generation of other documents (such as cross references, data dictionaries, and lists of items yet to be completed) is possible from ADL designs. Detailed designs, as described above, should be executable, if inefficiently. Rapid prototyping at higher levels of abstraction of design is perhaps beyond the present state of the art.

It should be possible to inquire about the consistency among the several levels of architectural designs, and detailed designs. The design language should have a sound enough foundation that it is possible to state the consistency problem, limited to the functionality ignoring other aspects such as real-time performance, as a purely mathematical logic statement to be proven. We do not expect these to be routinely proven either by humans or automated systems.

[1] Work supported by IBM Shared University Research Program.

[2] Names are alphabetically listed.

2.5 Ada Orientation

Beyond the expressiveness of Pascal-class of programming languages. Ada has the following notions that are so significant that every design language claiming to be "Ada oriented" must support them at the design level: packages, generics, exception handling, and tasks.

It should be clear, because of the context of this report, that we are considering design languages conducive to von Neumann machine architectures, and to the imperative style of programming. These two qualities of the primitives of implementation are well-captured by the Pascal-class of programming languages. Assuming that a design language supports the Pascal-class, the Ada orientation of it can be judged from how well supported are Ada specific constructs such as exceptions, generics, tasks, and packages.

3 Capsule Descriptions of a few ADLs

This section is a compendium of the design languages we studied. For each language, we give an objective overview of the particular language. and offer a subjective opinion of it.

3.1 ADA/SDP

ADA/SDP is based on a design tool called the Software Development Processor(SDP) developed by Linden [11] at UCLA. The language used for software designs in this tool came to be known as SDP. ADA/SDP is an extension of this language to allow use of Ada constructs like packages, generics, tasks, etc. and was developed at Mayda Software Engineering, Israel [1]. The major goal of this language has been to use programming language constructs with relaxed syntax thus increasing ease of expression and readability but at the same time providing some degree of formality.

A design in ADA/SDP is composed of modules which are subprograms, packages, tasks and their bodies. Module specification is always separated from the body by the use of the SEPARATE construct. Its use is similar to that in Ada except that they do not imply separate compilation units but are used as means to defer implementation details. This allows the designer to describe the software architecture first, by specifying the modules, the external modules that they use, and their interaction and then give their implementation in the corresponding bodies.

ADA/SDP allows stylized English text(pseudo code) at any place where an Ada statement or expression can appear. If pseudo code is written in a restricted way, some basic consistency checks of the design become possible, e.g. procedure interface, parameter types, etc.. An unparameterized item in ADA/SDP can consist of one or more words, e.g.

```
set of words

input file
```

are valid names for items. Parameterized items like subprograms, packages, tasks and generics, can consist of one or more words with the parameters embedded in the name of the item. Formal parameters are enclosed within "$", e.g. line 9 in Figure 1 is the header of a procedure which has two formal parameters - **word** and **input file**. Any text in the design description which matches the words other than the formal parameters is considered an application of the procedure. Types of the parameters can also be specified in lines following the procedure header declaration as shown in lines 9 - 12 in Figure 1.

An application of a subprogram is performed by writing the name of the subprogram with the formal parameters replaced by relevant text representing parameter values, e.g.

```
get a name from the source file
```

is a call to the procedure declared as above with **name** and **source file** as the actual values. As mentioned earlier, argument values can consist of one or more words.

```
0
.  text name intersection
.       This procedure takes as input two files with
.       names and produces as output another file which
.       consists of names which occur in both the files.
5  end text
.
.  separate(name intersection.name intersect file1 with
.           file3.build name set from file )
.  procedure get a $word$ from the $input file$
10         $word : string$
.          $input file : file type$
.
.     function $character$ is a letter return boolean
.
15 begin
.  end get a $word$ from the $input file$
```

Figure 1: Ada/SDP

ADA/SDP allows the use of text segments to present additional information about any design module. They are enclosed within keywords `text` and `endtext` and usually appear at the beginning of a module description as shown in lines 1 - 5 in Figure 1.

ADA/SDP simplifies Ada syntax for ease of expression. A number of visibility rules have been simplified. ADA/SDP does not have the USE construct of Ada and treats the WITH construct as being equivalent to it. The name of an entry is visible if the task name is visible. It does not require the body stub for a packages visible subprograms even though the bodies are defined SEPARATE.

ADA/SDP allows incompleteness in designs. Undeclared items can be used, type declarations can be omitted, module bodies can be absent, etc., e.g. lines 7 - 16 in Figure 1 form an incomplete procedure declaration.

The ADA/SDP processor can generate various documentations, like listings of design modules, call tree, data dictionary and cross-reference tables.

3.2 Anna

Anna (ANNotated Ada) is a language extension made to Ada to provide a means for formally specifying the intended behavior of Ada programs and support the various phases - specification, design, implementation, verification, testing, debugging, and maintenance - of their life cycle [13]. The design of Anna was initiated in 1980 by Bernd Krieg-Bruckner and David Luckham [10]. The version described here is an improvement on the 1980 version and was undertaken over the period 1981 - 1984 as part of the research project of the Program Analysis and Verification Group at Stanford University. It corresponds to the Ada language released under MIL-STD 1815A [6].

An Anna program consists of an Ada program interspersed with "formal comments", which constitute the actual Anna text. There are two types of "formal comments":

1. annotations with a leading "--|" consist of sequences of logical clauses expressing constraints over some program state components, the values of a data type, or the values of a variable of the underlying Ada text, and

2. virtual text with a leading "--:" defines a variable, a data type, or a function for use in the annotations.

With respect to the underlying Ada text, and except for exception annotations, "formal comments" must be side effect-free and compatible with the lexical and scope rules; that is, if the leading comment start sequence is removed from a "formal comment", the underlying Ada text should be syntactically correct in its context. The scope of a "formal comment" is that of the entity it annotates. Stand-alone

```
0
.  --|        I < BUFFERSIZE and not End_Of_File(F);
            while ISLETTER(C) loop
.              I := I + 1;
.              BUFFER[I] := C;
5  --|         IN I > BUFFERSIZE => raise OVERFLOW;
.              EXIT WHEN End_Of_File(F);
.              Get(F, C);
.           end loop;
```

Figure 2: Loop invariant annotation in Anna

"formal comments" have the scope that they would have if considered as part of the underlying Ada text.

The semantics of annotations is defined by means of a transformation that reduces Anna text to Anna kernel, a subset of Anna in which most annotations are Hoare assertions. For annotations that are not reduced to assertions, a system of axioms and proof rules based on "weak" logic makes it possible to perform formal proofs [14].

Data Type and Variable Annotation

The constraints expressed by a type annctation apply to all variables of that type declared in the scope of the annotation. Such an annotation immediately follows the type declaration (see below). A variable can be annotated the same way.

```
      type NAME is Access String;
 --|  where
 --|   for all N : NAME; I : positive range 1 .. length(N) =>
 --|   N.ALL /= "" and ISLETTER(N[I]);
```

Statement Annotation

A statement annotation constrains the values of the state components referenced by the statement. The scope of such an annotation is limited to the statement. The constraints are applicable before and/or after the execution of the annotated statement depending on whether they are modified by IN or OUT. Loop invariants are expressed in this manner (see line 1 in Figure 2).

Function and Procedure Annotation

A function annotation appears immediately after the interface specification of a function. It expresses constraints on the parameters and/or return value. Procedures can be similarly annotated.

Package and Generic Annotation

All the annotations described above and their scope and lexical rules are applicable to package interface specifications, package body definitions, and formal parameters of generics. In addition, axiomatic annotations only apply to packages and generics.

An Anna package is viewed as a composite object of some new type. Each package, and each instantiation of a generic package introduces a new type. The state of a package is strictly composed of the values of objects local to the package body. Axioms in the private part of the interface specification or in the body of a package constrain the state of the package. In the visible part of a package axioms are promises, which client programs can assume to hold. An example of axiomatic annotation is given in Figure 3. The annotations of a generic also apply to its instances. In addition, any constraints applied to an actual parameter before the instantiation of a generic also apply to the instance.

Exception annotation

The purpose of these annotations is to specify conditions under which a handler expects an exception, an exception is raised locally, or conditions under which a subprogram must propagate an exception. For instance, the annotation at line 5 of Figure 2 expresses that whenever execution reaches the preceding statement, if the condition "I > BUFFERSIZE" is true, the exception "OVERFLOW" must be propagated.

Annotations constitute a set primitives which when combined with the Ada private type facility makes Anna a powerful language for specifying abstract data types by the algebraic method. Anna may be the basis of a program verification and debugging environment provided that the appropriate tools are constructed to make use of its concepts such as package states and annotations. An Anna specification can be executed to check that the design meets it. It can also be run against the Ada program to detect departures from its expected behavior.

```
package SYNTAB is
  ...

--| axiom
--| for all SS : SYNTAB'TYPE; S, T : STRING; I : TOKEN =>
--|     SYNTAB'INITIAL[LEAVEBLOCK]  = SYNTAB'INITIAL,
--|     SYNTAB'INITIAL.IN_BLOCK(S)  = FALSE,
--|     SS[ENTERBLOCK; LEAVEBLOCK]  = SS,
--|     SS[ENTERBLOCK].IN_BLOCK(S)  = FALSE,
--|     SS[ENTERBLOCK].LOOKUP(S)  = SS.LOOKUP(S),
--|     SS[INSERT(S, I); LEAVEBLOCK] = SS[LEAVEBLOCK],
--|     SS[INSERT(S, I)].IN_BLOCK(T)  =
--|      if S = T then TRUE else SS.IN_BLOCK(T) end if,
--|     SS[INSERT(S, I)].LOOKUP(T) =
--|      if S = T then I else SS.LOOKUP(T) end if;

end SYNTAB;
```

Figure 3: Anna axiomatic description

3.3 Arcturus PDL

Arcturus is an Ada-based programming environment developed at the University of California at Irvine. One of the components of this environment is an interpreter which supports an Ada-based PDL and a Rapid Prototyping Language(RPL) [17].

The Arcturus PDL uses Ada constructs and also allows deferring detailed design decisions via text and descriptions. The text is enclosed within braces. They are treated as PDL comments by the interpreter. Programs with the PDL text can be interpreted normally as long as control does not reach any PDL text. When it does, a **break procedure** is invoked and execution is not continued. The RPL provides a macro expansion facility by which a PDL text can be refined to Ada code and/or other PDL texts, thus allowing stepwise refinements and multiple levels of design. A program becomes fully executable when all such PDL texts can be expanded to Ada code.

The Ada constructs which can be replaced by text are,

- **declarations** e.g., { (name_set) is a tree of names }
- **expressions** e.g., { maximum of (a) and (c) }
- **names** e.g., { array of words }
- **statements** e.g., { insert node (a) to tree (t) }
- **types** e.g., type symtab is { table of symbols }

The PDL text is a sequence of valid Ada identifiers. Any of the reserved words of Ada can be used, except **begin, end** and **declare**. It can also contain parenthesized lists of Ada expressions separated by commas. As can be seen from the examples. PDL texts can be combined with any of the Ada constructs (refer to line in example)

The macro definition of RPL is used to refine a PDL text to Ada code and/or other PDL texts. A PDL text so used is called a **calling form**. The text of the calling form identifies the macro which is to be used for expansion. The parenthesized expression list form the variable

```
0
.    function LOOKUP(S : STRING) return TOKEN is
.
.        I : INDEX_RANGE;
.
5      begin
.       { check symbol table in reverse from (INDEX)
.          and for each entry do
.          begin
.            { if symbol is (S) then return token };
10         end
.       };
.       raise NOT_FOUND;
.    end LOOKUP;
.
```

Figure 4: Calling forms in Arcturus

```
-- Macro definitions for PDL calling forms.

package  symtab_macros is

macro { symbol info record } return typenode;
macro { storage of (maxsize) size for (element)
          records } return typenode;

end symtab_macros;

macro symtab_macros body is

macro { symbol info record } return typenode is
begin
        return gentype(
            "record" &
            "LEVEL  : NATURAL;" &
            "MEMBER : STRING;" &
            "TOK    : TOKEN;" &
        "end record;");
end;

macro { storage of (maxsize) size
          for (element) records }
return typenode is
begin
 return gentype("array (1 .. $maxsize) of $element");
end;
```

Figure 5: Arcturus macros

part of the calling form and is referred to as the argument list. A calling form can also contain an argument list of statements. A statement argument list is delimited by the reserved words **begin** and **end**, (see e.g. line 6 of Figure 4). Calling forms can also be nested as shown in line 9 of Figure 4.

Macros are syntactically similar to functions but unlike regular functions they are used to return only one type of value - a node in the parse tree (Ada programs are represented as parse trees in the interpreter). The type of nodes that a macro can return are declarations, types, expressions, statements and names - the Ada constructs that can be replaced by a calling form. The type of node determines whether a calling form which invoked the macro can be used legally, e.g. a declaration node is only valid where a declaration is valid in Ada. Functions like gendecl, genexpr, genname, genstmt and gentype are provided to construct the nodes.

Macro definitions are valid wherever a procedure or function is valid. The scope of a macro definition is determined by the scope in which it is defined. Packages of macros can be defined; e.g. Figure 5 uses a macro package definition for refining the calling forms used in the program. If the macro definitions need to share information among themselves, they can do so by variables declared in the package.

A macro expansion occurs when control reaches a calling form. The sub-tree returned by a macro is attached to the program parse tree and execution continues normally. Macros can also be expanded by the use of interpreter function **expand**. The use of expand results in actual text replacements in the program. The interpreter also provides a function **unexpand** which restores the original program text

The Arcturus PDL together with RPL provides a very convenient environment for design. Designs can be expressed at a very abstract level and can gradually be refined to capture a more detailed design with the use of the macros. At any stage it is possible to see how a higher level of design gets refined to a more detailed, lower level of design. It is possible to have multiple design representations - a different set of macro expansions can represent some other way of refining the design. Even though the idea of the design process is well captured, RPL's syntax is often cumbersome as can be seen from the examples. The mechanism of use of the macros is dependent on the internal representation of programs in the interpreter. Constructs like "refines to" is a more natural way of specifying the refinement process. Because of its Ada orientation, Arcturus PDL is definitely very useful as an Ada PDL.

3.4 BYRON

Byron is an Ada based PDL developed at Intermetrics, Inc. [8]. It was introduced in the year 1983. The Byron PDL consists of informal English language sentences(Byron constructs) interspersed with Ada code. Byron constructs appear as legal Ada comments to the compiler and so Byron is fully compilable by any Ada compiler. However, Byron constructs can be differentiated from regular Ada comments. Their general form is

```
--| <keyword> : <text>
```

The **<text>** part is made up of arbitrary English sentences and may span over more than one line, each line beginning with the Byron prefix "--|". The purpose of the text is to informally specify different parts of the design/code. Utilities/tools are provided in the environment which extract the text and produce documentation for the design/code according to certain documentation standards.

Since Byron does not support formal representations of the text, no type can directly be associated with the text. The **<keyword>** part of the Byron construct indicates the type of text that follows it. For the same reason, no checking can be done by the processor to verify whether the content of the text is meaningful under the specified keyword or not.

The keywords can be of two types -

1. **Byron flags**: The flags are used to guide the processor in formatting the document. There are flags which mark a block of code to be extracted for use in documentation by the Byron processor.

2. **Byron directives**: The directives are based on a classification of the different types of design information needed to specify a section of the code. They can be one of *Algorithm*, *Effects*, *Errors*, *Invariants*, *Modifies*, *Notes*, *Overview*, *References*, *Representation*, *Raises*, *Requires* and *Timing*. Figure 6 contains an [illegible] of some of the directives. With the help of these directives the designer can informally specify the relevant aspects of a subprogram. As suggested by the names of the keywords , there are directives to indicate the algorithm of a subprogram, the variables it can modify, the exceptions it raises, its requirements, its data structure representations, etc. All Ada constructs - functions, procedures, packages, tasks, etc., can be described using the Byron directives.

```
package SYNTAB is
  ...
procedure INSERT(S : STRING; --| symbol to be inserted
              I : TOKEN   --| tokn of the symbol
              ); --| raises OVERFLOW

--| Overview
--| Inserts a new symbol in the symbol table with string
--| S and token information I and the current lex level.

--| Effects
--| If INDEX not equal to MAX then INDEX and TABLE(INDEX).

--| raises OVERFLOW if INDEX = MAX.

end SYNTAB;

package body SYNTAB is
--| Overview
--| This package provides operations on a symbol
--| table. The table is represented by an array of
--| records. The operations can insert new symbol
--| and lookup the symbol table for a symbol
--| at different lexical levels.

    type ELEM is record
                LEVEL  : NATURAL;
                MEMBER : STRING;
                TOK    : TOKEN;
               end record;
    type STORE is array (1 .. MAX) of ELEM;
    subtype INDEX_RANGE is NATURAL range 0 .. MAX;
    TABLE    : STORE;
    LEXLEVEL : NATURAL := 0;
    INDEX    : INDEX_RANGE := 0;

    procedure INSERT(S :STRING; I : TOKEN) is
--| Algorithm
--| If current index equal to MAX then the exception
--| OVERFLOW is raised else increments the index and
--| assigns the indexed position in the table with
--| LEXLEVEL, symbol string and the token value.
    begin
        if INDEX = MAX then raise OVERFLOW;
        else
           INDEX := INDEX + 1;
           TABLE(INDEX) := ELEM'(LEXLEVEL, S, I);
        end if;
    end INSERT;

 end SYNTAB;
```

Figure 6: Byron example

3.5 PDL/81

PDL/81 is from the group [3] that brought attention to the concept of "design language" in the mid 70's. It is designed for the production of structured designs in a top-down manner. PDL/81 [4] is a "pidgin" language in that it uses stylized English language and the constructs of a structured programming language, such as Ada.

PDL/81 is not presented by its vendor as a "design language", but rather as a set of tools to help design. Input of the PDL/81 processor consists of several "segments" which are: TEXT, SPECIFICATION, PROCEDURE, FUNCTION, and TASK BODY segments. The output is a "design document." Each segment begins with a line such as

```
%keyword descriptive id
```

A descriptive id usually contains several words. Here is a list of keywords: TEXT, PROCEDURE, FUNCTION, TASK BODY, and SPECIFICATION.

Text

An unformatted text segment is introduced by the command

```
%TEXT Introduction
The program processes input word stream from
two files that...
```

where "Introduction" is the title of the text "segment" and the body of the segment is describing something.

Specification

SPECIFICATION segment is a collection of named data items and their descriptions. E.g.

```
%SPECIFICATION data definition

word is a sequence of characters
text is a sequence of words
```

In the above, "data definition" is the name of this segment. The first word in each line of the body, **word** and **text**, are explicitly declared to be data items. The rest of the line is taken as commentary, and can, in general, be an arbitrary sequence of characters for description.

Procedure, Function, and Task Body

All these segments are used to express the functional decomposition of a design. A line such as

```
%PROCEDURE form_a_set (set_S, file_F)
```

introduces a procedure segment named "form_a_set". Its body can be elaborated, as e.g. in

```
create an empty set
WHILE get_a_word(word_W,file_F) is success
  LOOP
    add_word_to_set(word_W,set_S)
  END LOOP
close (file_F)
```

The statements included in the body of a procedure segment are of two kinds: "control statements" and "descriptive statements." Control statements are Ada control structures with a boolean expression replaced by a sequence of words. Descriptive statements are arbitrary sequences of words. However, words that begin a statement have special meaning, e.g., SET x to 1, means assign value 1 to data item x. Apart from this, no formal syntax is implied in the statements. So one can write, in English prose, a description of the intended action.

3.6 PDL/Ada

The Process Design Language/Ada-2 (PDL/Ada-2) [2] is a product of the Ada Steering Group of IBM Federal Systems Division. The current version is an upgrade of PDL/Ada developed by the same group in 1981 [16]. The original version of PDL/Ada was based on the ideas of PDL by Linger [12].

PDL/Ada-2 supports full Ada. The features that PDL/Ada-2 adds to those available in Ada are "function abstraction" and "data abstraction." Design expressions are presented as Ada comments and thus PDL/Ada-2 is fully compilable by an Ada compiler. Ada control structures, mathematical notations, and English statements can be used to present designs. Designs in PDL/Ada-2 are represented at various levels of abstraction. Each abstract level is then refined to lower levels of abstractions or concrete implementations.

Function Abstraction

Function abstractions in PDL/Ada-2 are expressions of behavior that are expressed using a special form of Ada comment. They are used to specify the behavior of a segment of design, a procedure call, a single control structure, a procedure, or an entire system or subsystem. Function abstractions are presented in PDL/Ada-2 in the following ways:

1. Behavior Specification

Behavior specifications in PDL/Ada-2 are surrounded by either <, > or [,]. They specify the behavior (or function) of a design component. They appear as

```
-- [ Behavior Specification
--    ...                         ]

-- < Behavior Specification
--    ...                         >
```

The text that goes between the brackets can be one of the following types of statements:

- **Simple assignment statement**: shows the change in value of a single variable.
- **Multiple assignment statement**: shows the changes in value of multiple variables.
- **Concurrent assignment statement**: indicates multiple data transformations, as in:

  ```
  --< ((SYMTAB.LEXLEVEL := 0), (SYMTAB.INDEX := 0)) >;
  ```

- **Conditional assignment statement**: represents a sequential guarded command, e.g.,

  ```
  --< SYMTAB = {} -> IDENTITY
  --   | TRUE -> [ LASTB := (I in DOMAIN(SYMTAB) :
  --               (for all J in DOMAIN(SYMTAB) : I >= J))
  --               in SYMTAB(LASTB) := UNDEFINED ]            >
  ;
  ```

  Expressions in the statements represent a value to be assigned to the corresponding variable. Expressions can be written as Ada expressions, English statements or in mathematical notations.

- **Identity statement**: indicates that no data transformations take place; no variables change value (see the first line of the code for the conditional assignment statement above).
- **Textual assignment statement**: represents assignment using English text or mathematical notations, i.e.,

  ```
  --< Return value := the intersection of N1 and N2 >
  ```

When a behavior specification represents the lowest level of stepwise refinement of a function abstraction, it is terminated by a semicolon. Such a specification can then be refined to a concrete implementation in Ada. When this is done, the semicolon from the behavior specification statement is converted into an "is" keyword followed by the Ada statements, to indicate the "refinement." For example:

```
--< swap x and y >;
```

becomes

```
--< swap x and y > is
t := x;
x := y;
y := t;
```

2. Intended Functions

Intended functions are English descriptions used prior to statements beginning with the keywords **procedure, function, or accept**. It states the intended purpose of a design unit, e.g.,

```
--< Return a set which is the intersection of two sets. >
function INTERSECTION(N1, N2 : NAME_SET) return NAME_SET
```

Data Abstraction

Data abstractions are expressed in PDL/Ada-2 by modules. PDL/Ada-2 uses Ada packages to construct two kinds of Data abstraction modules: Abstract Data Types (ADTs) and Abstract Data Objects (ADOs). In PDL/Ada-2 ADT and ADO packages include additional information, in the form of special comments, to express aspects of the data abstraction.

The *module specification* includes two parts:

1. **State Section:** The *Model* describes the name of the abstract model. The *Constraint* specifies the restriction of the abstract data model. The *Initial* gives the initialization of the data model.
2. **Transition Section**: It consists of subprogram specifications whose behavior specifications are expressed in terms of operations on the abstract data models.

The body of the module includes two parts:

1. **State Section**: This appears in the package body for ADOs and in the private part of the package specification for ADTs.

   The *Representation* is the description of lower level data abstractions or primitive data types. The *Constraint* specifies the restrictions on the data representation. The *Initial* describes initialization of the data representation. The *Mapping* shows the relationship of the data representation to the data model.
2. **Transition Section**: In this section, the behavior specifications of the data operations in the module specification are restated in terms of the data representation.

The stylized comments in the specification part provides information about the data model — the properties of and the operations on the data model. The module body (package body) contains the details of the data model or the implementations of the module specification.

In designs, the keyword CONDITION is used to indicate the use of an abstract predicate. It is followed by the abstract predicate expressed as an Ada comment:

```
if
 CONDITION --< SYMTAB.TABLE(I).LEVEL < SYMTAB.LEXLEVEL >;
then
 return FALSE;
elsif
 CONDITION --< SYMTAB.TABLE(I).MEMBER = S >;
then
 return TRUE;
end if;
```

PDL/Ada-2 has a "Design Support Package" that provides predefined abstract types, such as **set, sequence, stack,** and **queue**; TBD type-definition, e.g. TBD_TYPE; pre-defined boolean data objects, e.g. CONDITION, CD; and dummy statement, e.g ST (sequence of statements), procedure_TBD.

4 Table of Comparisons

Section 2 presented a general discussion of what we expect of an ADL. In this section we assign ratings to the ADLs under specific criteria. We present our interpretation of each criterion and also explain how we will proceed to rate a language under that criterion. The rates are given on a scale of 1 to 10. A 0 rating is given if the language does not support a criterion in any way.

4.1 Design Expressibility

We examine each design language to see how it supports:

1. **Describing data structures**: formal definition, constraints of values, operation allowed on objects.
2. **Expressing algorithm designs**: precise, formal, and rigorous.
3. **Describing the behavior of components**: execution order, data flow, exceptions, and error conditions.

This criterion evaluates the expressive power of each language. Some languages, e.g. PDL/81, tend to be informal and flexible enough to permit the use of natural language and will thus get high points in expressibility but low points in formal and precise algorithm design.

4.2 Design Hierarchy

Three criteria account for the rating assigned to an ADL under this column:

1. the degree of support the language provides for organizing the static structure of a design in hierarchical manner,
2. the expressive power of the ADL in describing the "uses" relationship [Parnas 79] among design components, and
3. the levels of abstractions attainable in using the ADL.

An ADL that uses nothing more than Ada constructs is given a zero rating, even though some abstraction is in this sense achievable with Ada. The higher the level of the data types and control constructs provided by an ADL than those of Ada, the higher its rating, provided it also meets the other criteria.

4.3 Design Mapping

The rating under this column is based on the capabilities of the ADL in relating designs. If an ADL provides a way of relating designs, a minimum of 1 point is given to it. The remaining points are awarded based on how checkable the validity of the relations is, because of the nature of the ADL. For instance, an ADL that relates designs by means of English comments is assigned a total rating of 1; this also holds for formal ADL's relating designs in such a way that no checking procedure is likely to be devised. The highest ratings are given to ADL's that relate designs in some way for which a checking procedure exists, or for which we are sure of the possibility of devising one.

4.4 Rapid Prototyping

A prototype is used to assist in the evaluation of functional requirements. There are two major approaches to rapid prototyping: reusable code and executable specifications. If the design language is an executable specification then it will be graded no less than 5. Reusing code to achieve prototyping will compete by its flexibility and reusability.

4.5 Consistency Checking

This relates to the feasibility of writing a tool to perform consistency checks of the design written in the language under consideration. The consistency checks in question are:

- module interface checks,
  - right number of arguments used for an application of a defined module,
  - correct types of arguments,
- consistent use of declared variables with respect to their types,
- consistent use of values returned by functions.

Languages for which such tools can be built, are rated according to the extent to which such checkings can be done correctly. Consistency checks are possible with languages with formal syntax and semantics. For languages like Byron, where a design is described by informal English language text, no consistency checker can be built and should thus be given a rating of 0 under this heading.

Evaluation Table

Criterion	Anna	SDP	PDL/81	Byron	PDL/Ada	Arcturus
Design expressibility						
1. data structure	4	1	1	1	2	1
2. algorithms	2	1	1	1	1	3
3. Behavior	6	0	0	0	2	0
Design hierarchy	2	1	1	0	1	4
Design mapping	1	1	1	1	2	3
Prototyping	6	0	0	0	0	5
Consistency checking	8	3	1	0	2	4
Formal syntax	8	2	1	1	4	5
Formal semantics	8	1	0	1	4	3
Completeness checkability	1	1	1	1	1	1
Understandability	1	3	4	7	5	4
Constructs related to Ada	7	1	2	1	2	1

4.6 Formal Syntax and Semantics

Perhaps because of the requirements of DoD-STD-2167, most of the Ada design languages express their legitimate syntactic constructs, as well as *design comments*, in the double-hyphen Ada comments. By definition, we consider any language permitting arbitrary sequences of "words", except in these so-called design comments, to have no formal syntax.

4.7 Completeness Checkability

This relates to the feasibility of the tool to examine the design objects for the detection of deferred design (*incompleteness*). Languages for which completeness is checkable, are rated according to the checkings that can be done correctly.

4.8 Understandibility

The understandibility criterion relates to the ease of understanding the object that is being described as opposed to understanding the language itself. Understanding an object is often influenced by the style of presentation, and the complexity of the object itself. We do not consider these issues in rating a design language. Readability of the language plays a very important role in the understanding of designs. Since there are no metrics by which this can be measured, we do [illegible] that our considerations are totally subjective.

4.9 Constructs Related to Ada

In order to make the transition from the design phase to the implementation easier, it is often helpful to have DLs support constructs defined in the target implementation language. The DLs need not use the constructs implicitly but should have well defined mappings to them. The ease with which such mappings can be defined is the criterion for rating DLs under this heading. For DLs which use Ada constructs directly, this mapping becomes trivial. We thus consider language constructs over and beyond Ada constructs in rating the language under this criterion.

5 Conclusions

A great deal of effort has been dedicated to the design of ADLs since the Ada language was released in 1980. Still, most ADLs are nothing but a reincarnation of the earlier idea of using pseudo-code to design software. Such ADLs are informal in the sense that they do not have a formally defined syntax, less do they have formally defined semantics.

It seems that the goal of these languages was to make it possible to generate various documents about the design being constructed rather than to directly support its construction. Tools for producing *cross-references*, call-trees, data dictionaries and so forth are associated with almost every ADL. On the other hand, almost none of these ADLs provides direct support for the descriptions of designs of Ada programs: direct support for Ada notions such as generics, derived types, packages, and tasks is skimpy.

However, from this crowd of ADLs, Anna stands out with its formally defined syntax and rigorously described semantics. Its foundation is in mathematical logic and discrete mathematics. It is certainly true that this ADL is not readily accessible by the average programmer without considerable training; but, tools based on formal languages are more likely to be capable of directly assisting software designers in their effort of constructing designs. In contrast, not much processing can be performed to actually assist informal ADL users. If informal ADLs are being so widely used for the sake of human readability, ADL designers must find a means of providing both readability and formalism for these languages.

References

[1] Ada/SDP Introductory Guide, Mayda Software Engineering, Israel, (1985).

[2] Ada Steering Group, "Ada-Based Process Design Language *(PDL/Ada-2)*," *IBM Federal System Division*, 6600 Rockledge drive, Bethesda, Maryland 20817, (December 1986).

[3] Caine, S. and Gordon, K., "PDL - A Tool for Software Design," in *Proceedings of the National Computer Conference*, AFIPS Press, pp. 217-276, (1975).

[4] CFG Inc., "PDL/81 A tool for Software Design, An Introduction," Caine, Farber & Gordon, Inc., 2nd Edition Pasadena, California, (1985).

[5] Computer Technology Associates and Advanced Software Methods, "Survey of Ada-based PDLs," TP-598, Contract N00163-84C-0300, Naval Avionics Center, Indianapolis, Indiana, pp. vi, (January 1985).

[6] U.S. Department of Defense, Ada Joint Program Office, "Ada Language Reference Manual," MIL-STD-1815A, (January 1983).

[7] U.S. Department of Defense, Military Standard, Defense System Software Development, DoD-STD-2167, pp. 28, (4 June 1985).

[8] Gordon, M., "The Byron Program Design Language," *Ada Letters* Vol. 2, No. 4, pp. 76-83, (1983).

[9] IEEE, "Standard: 990-1987 Ada as a Design Language," IEEE Ada as a PDL Working Group Recommended Practice, *IEEE Computer Society*, (1987).

[10] Krieg-Bruckner, B. and Luckham, D.C., "Anna: Towards a Language for Annotating Ada Programs," *Proceedings of the ACM-SIGPlan Symposium on the Ada Programming Language*, Vol. 15, No. 11, pp. 128-138, (November 1980).

[11] Linden, N., "Software Design Processor: A Tool for Program Design," M.S. Thesis, UCLA, (June 1975).

[12] Linger, R. C., H. D. Mills, B. I. Witt, "Structured Programming: Theory and Practice," Addison-Wesley, (1979).

[13] Luckham, D.C., Von Henke, F.W., Krieg-Bruckner, B., and Owe, O., "Anna a Language for Annotating Ada Programs," Technical Report No. CSL-84-261, Stanford University, California, (July 1984).

[14] Owe, O., Krieg-Bruckner, B., Von Henke, F.W., and Luckham, D.C., "Axiomatic Semantics of Anna," *Program Analysis and Verification Group Report*, Stanford University, California, (1984).

[15] Parnas, D.L., "Designing Sotware for Ease of Extension and Contraction," *IEEE Transaction on Software Engineering*, Vol. SE-5, No. 2, pp. 128-137, (March 1979).

[16] Sammet, J.E., Waugh, D.W., and Reiter Jr, R.W., "PDL/Ada - A Design Language Based on Ada," *Ada Letters*, Vol. 2, No. 3, pp. 19-31, (November/December 1982).

[17] Tadman, F.P., "The Arcturus Programming Environment, Program Design and Rapid Prototyping Language," Programming Environment Project University of California, (25 January 1984).

FORMAL CONCURRENT TASKING PARADIGMS IN THE DESIGN OF ADA PROGRAMS

Ronald J. Leach
Darlene Bond

Department of Systems & Computer Science
School of Engineering
Howard University
Washington, D.C. 20059

ABSTRACT

A major feature in the design of Ada was the high level support of concurrent tasks. Concurrent tasking is an essential feature of embedded systems in most environments. In this paper we examine the state of Ada education in the support of concurrent tasking. The tasking examples are compared to formal models in C.A.R. Hoare's CSP (Communicating Sequential Processes) system. The resulting information is compared with actual use of tasking programs in the Ada literature and in industry and government. Particular attention is paid to the treatment of non-determinism in tasking programs and in formal models.

INTRODUCTION

In order to maximize the effectiveness and efficiency of a program, a programmer must begin the program development with a good program design structure. Programs are made up of several types of building blocks. Programs without concurrent execution of tasks use the standard sequential building blocks of procedures, functions, and modules. Programs involving concurrent execution use these building blocks and the additional block of a task which is usually a collection of the sequential building blocks. The use of concurrent tasking in programs greatly increases the potential for error in programs and thus causes great difficulty during all phases of the software life cycle. Errors which occur at many phases of the software life cycle and costs which increase exponentially are major features of the "software crisis". It is clear that the current "software crisis" will get even worse since most of the existing problems have been with systems which do not involve much concurrent execution.

Abstraction and information hiding are major techniques of software engineering that are used to address some of the problems with software. Indeed, the relative ease in which information hiding and abstraction of data are implemented in the Ada language is a major reason for the success of Ada. It is clear from the success of these language features that there is a need for formalism in the area of concurrent programming in Ada.

Perhaps the most common paradigm for design of programs involving concurrent tasks is C. A. R. Hoare's Communicating Sequential Processes (CSP). It is reasonable to ask if Hoare's abstract models of CSP involving concurrency are applicable and effective tools in program design. Ada was originally intended for use with embedded systems and concurrent tasking and to incorporate principles of good software engineering; it is appropriate at this point to examine how these two ideas work together in practice. This research was conducted to see if the current state of use of abstract models of Ada programs involving concurrent tasking is sufficiently well-understood to be used in providing a basis for Ada program design. Thus this research represents an assessment of how well the use of tasking and formal models is supported in the existing Ada educational community.

The first step in approaching this problem was to collect data. The data was initially collected from the published literature of Ada programs including textbooks, lecture notes, and conference proceedings. We chose 17 texts from the library; the selection criterion was actually having the book on the shelf and not in circulation at the time that the data was gathered. We feel that this is a representative sample of the use of tasking in the existing Ada textbook literature. Programs which involved tasking were extracted and examined to see ered. We feel that this is a representative sample of the use of tasking in the existing Ada textbook literature. Programs which involved tasking were extracted and examined to see which, if any, of Hoare's models could be applied to the programmers' method of executing the task or tasks. The results were tabulated to see which models were applied in these programs.

The textbooks examined fall into two categories: limited amounts of tasking (including none at all) and considerable emphasis. A total of 819 programs from all textbooks were examined, with only 114 or 13.9% having any concurrent tasks. We note that most of the programs involving tasking (36) were found in a single reference [7]. Of the programming samples obtained from textbooks in the first category, there were only 32 programs involving tasking out of a total of 730 or 4.3%. The P||Q model of concurrent execution of tasks with no communication between the tasks was found most often, with a total of 14 instances, of which 12 involved only two tasks. The repetition of tasks, which is denoted abstractly as *P, was the next most frequently found, with seven instances. The next most frequent model occurring is the P;Q model in which the tasks actually are executed in order, a total of 6 instances. Hoare distinguishes 29 distinct models for tasking involving two tasks; only 8 of them or 27.5% are represented in the texts. In table 1 below, we summarize our search of the the textbook literature, some examples of student programs, and sample programs that are available in the non-textbook Ada literature. Note that we show the number of tasks in each example and therefore do not quite agree with all of Hoare's categories, since Hoare only lists the possibilities for the execution of *two* concurrent tasks in his explicit listing of possibilities.

The textbooks [5] and [7] had much more emphasis on concurrent programming as the titles "Concurrent Programming in Ada" and "Parallel Programming in ANSI Standard Ada" would indicate. There were a total of 89 programs presented with tasking evident in 42 or 51.7%. Here the range of programs is much wider including examples of (P||Q)*, $$(P sub 1 ||P sub 2 ..||P sub n)* $$, P||Q with Q of the form R||S;T and several other models.

The next set of data was obtained from student programs. The intention here was to measure the level in which tasking is used in such programs. A preliminary experiment involving the examination of 12 student programs using concurrency indicated that the sequential execution of tasks predominated, with 8 uses of the P;Q model of sequential non-communicating tasks, 2 with parallel execution of non-communicating tasks (P||Q), one with the (*(*(*P;Q))) model of repeated sequential tasks, and one with the P;*Q model of task followed by repetition of a sequential task. Some of the programs obtained from students followed the P;Q model which describes the execution of two processes or tasks which are executed sequentially. Some observations about the difficulties encountered by students in the development and execution of these programs was made in [14].

An additional data set was obtained from the existing published non-textbook literature. This data was obtained from the newsletter AdaLetters (including its predecessor), proceedings of several Ada conferences, the Journal of Pascal, Ada, and Modula-2, materials from a variety of Ada short courses, and the Ada Repository. Again in this case, few of the sample programs supported the more complex models.

The most common example of Ada tasking programs was the consumer-producer problem which was presented in various forms. Many texts, especially [7], gave several different solutions to this problem. In some instances, there were two relatively different coding solutions to the same abstract model, even though the two models appeared to have the same CSP representation. We intend to pursue this subject in future work.

The remaining data was collected from a small set of programs actually used in industry and government. Some of these programs make elaborate and extensive use of tasking while of course others do not. The data collected is incomplete at this point because of the difficulty in obtaining samples of actual proprietary code. We do not expect that this data will ever be complete or that it will represent the precise percentages of use of Ada tasking in Ada programs. Instead, we consider it as an example of how Ada tasking paradigms are used in a few hopefully representative Ada applications.

TABLE 1: READILY AVAILABLE INFORMATION ON TASKING. NOTE THAT SOME OF THE DATA COULD ALSO BE CONSIDERED AS MORE COMPLICATED CSP MODELS

CSP MODEL	NUMBER OF OCCURRENCES			
	BOOKS	BOOKS WITH TASKING	STUDENT PROGRAMS	LITERATURE
P||Q	12	18	2	8
P[] Q	1			
*P	7			
P;Q	2	2	7	
b*P		1		
P//Q	1			3
x:A->P(x)	1			
A->P	1			2
*P;Q	1			
P||Q||R	2	8		3
P;Q;R	1			
P//Q||R	1	1		
P[]Q[]R	1			
P||Q||R||S	1	2		4
P;*Q	1			
(((P;Q)*)*)*			1	
(P||Q||R)*		1		
(P||Q)*		3		
P1 .. Pn		2		
(P1 .. Pn)*		1		
((Q//P1) ..||(Q//Pn))*		2		
TIMED TASKS				2

SUMMARY AND CONCLUSION

It is clear that the quality of information available to beginning and intermediate Ada programmers and designers about tasking is quite limited and does not address the full range of potential tasking uses. The actual problem is much worse than this because Hoare's CSP models do not allow for time constraints such as delays and fixed waits. Such factors are critically important in situations such as the FAA control system or indeed in any system that must perform in real time.

It is well-known that even experienced programmers have considerable difficulty in writing programs which involve any degree of concurrency. We recommend the following solutions.

1. At the preliminary level of education; that is, in the undergraduate and graduate programs of colleges and universities, the amount of instruction in concurrent programming must be increased. This instruction should be done over a variety of courses so that students see these ideas in a number of contexts.

2. Textbooks in the language Ada must include a wider variety of tasking programs including more of Hoare's CSP models. While the amount of tasking information need not be as much as in [7], it must be increased in order to make sophisticated knowledge of Ada tasking available to as many students as possible.

3. Continuing education for the professional should include a comprehensive study of tasking in Ada. This is not appropriate for the first introduction, which should be limited to the fundamental features of the language and Ada software engineering with only a brief introduction to tasking. Second courses should give views of many abstract models of tasking by means of many

different examples. We note that this is being done at Pennsylvania State University (Capitol Campus) and at Computer Science Corporation (Moorestown).

4. In the absence of high quality educational opportunities or having existing personnel already well trained in Ada tasking, management must choose between using special expertise from outside the organization and restricting the tasking to the simple models supported by most of the existing texts.

Acknowledgement

Research of both of the authors of this paper was partially supported by the U.S. Army Research Office under grant number DAAL-03-86-G-0085.

REFERENCES

1. Amoroso, S. and G. Ingargiola, *Ada : An Introduction to Program Design and Coding*, Pittman , Boston, 1985.

2. Ausnit, C. et al, *Ada in Practice*, Springer-Verlag, New York, 1985.

3. Booch, G., *Software Engineering with Ada*, Benjamin Cummings, Menlo Park, California, 1983.

4. Buhr, R.J.A., *System Design with Ada*, Prentice-Hall, Englewood Cliffs, 1984.

5. Burns, A. *Concurrent Programming in Ada*, Cambridge University Press, Cambridge, England, 1985.

6. Caverly, P. and P. Goldstein, *Introduction to Ada: a Top-down Approach for Programmers*, Brooks/Cole, Monterrey, California, 1986.

7. Cherry, G., *Parallel Programming in ANSI Standard Ada*, Reston Publishing Co., Reston, Va, 1984.

8. Cohen, N., *Ada as a Second Language*, McGraw-Hill, New York, 1986.

9. Downes, V.A. and S.J. Goldsack, *Programming Embedded Systems with Ada*, Prentice-Hall, London, 1982.

10. Gehani, N., *Unix Ada Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

11. Haberman, A.N. and D.E. Perry, *Ada for Experienced Programmers*, Addison-Wesley, Reading, Massachusetts, 1983.

12. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

13. Katzan, H.Jr., *Invitation to Ada*, Petrocelli Books, New York, 1984.

14. Leach, R., *Experiences Teaching Concurrency in Ada*, AdaLetters, 1987.

15. Mohnkern, G.L. and B. Mohnkern, *Applied Ada*, Tab Professional and Reference Books, Blue Ridge Summit, Pennsylvania, 1986.

16. Pyle, I.C., *The Ada Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

17. Texel, P.P., *Introductory Ada: Packages for Programming*, Wadsworth, Belmont, California, 1986.

18. Wegner, P., *Programming with Ada: An Introduction by Means of Graduated Examples*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

19. AdaLetters, various issues 1983-1987.

20. Journal of Pascal, Ada, and Modula-2, various issues 1986-1987.

21. Proceedings of the First Annual National Conference on Ada Technology

22. Proceedings of the Second Annual National Conference on Ada Technology

23. Proceedings of the Third Annual National Conference on Ada Technology

24. Proceedings of the Fourth Annual National Conference on Ada Technology, Atlanta, GA, March 19-20, 1986.

25. Proceedings of the Joint Ada Conference, Washington, DC , March 16-19,1987.

Darlene Bond received a Bachelor of Science degree in Psychology from Howard University in 1983. She is a graduate research assistant in the School of Engineering's Systems and Computer Science Department at Howard and will receive a M.S. in Computer Science in May, 1988. Her professional interests include systems programming, computer graphics, software engineering, and artificial intelligence.

Ronald J. Leach is a Professor in the Department of Systems and Computer Science at Howard University. His research interests include software engineering, computer graphics and concurrent computing.

Ada Projects for an Advanced Course Which Emphasize Machine Specific Aspects of the Language

Claus P. Janota

ARL Penn State
P. O. Box 30
State College, PA 16804

ABSTRACT:

When teaching an advanced Ada course it is a challenge to assign student projects which are meaningful but which can be accomplished in the limited time available. The instructor needs to take into account the range, and sometimes conflicting, requirements of a mix of students with various backgrounds and educational objectives.

The projects described in this paper are some which the students can choose to complete in about seven weeks. In all cases, several students work together to design and implement Ada software which emph sizes the advanced features of the language. Students present their chosen projects to the class initially, give a status report, and then participate in a structured walk through of the code near the end of the course. The other members of the class serve as the customers to critique the development and to provide feedback to the team members.

Intent of the Course

The advanced Ada course, as thought at The Pennsylvania State University, is the third of a series which comprise a Continuing Education certificate program in the Ada programming language. The course also carries college credits and is listed as a special topics offering in Computer Science. The first course in the series is one on Software Engineering Principles and the second one is an introduction to Ada which stresses the syntax and semantics of the language.

The students entering the advanced Ada course are therefore expected to be reasonably conversant with the structure and intent of Ada. And the design of the third course attempts to tie together the material presented in the prerequisite courses. The projects the students work on are intended to reinforce an appreciation of the major advantages of Ada; modularity, transportability, and flexibility. In fact, the projects also serve to sensitize the participants to the lack of uniformity in compiler treatment of low-level features and to things to be aware of in the application of Ada to "real" problems.

Students are set the goal of designing a demanding project and implementing that in a practical application. The projects are intended to be of sufficiently large scope that a single student could not complete the work in the time allotted.

Method

At the second lecture, high level specifications are presented for the various projects which the instructor offers as a list of possible candidates to work on. The list of suggested student undertakings is shown in Table 1. The fact that students will be expected to work in teams of two to four is explained and the rationale for that approach is detailed. Because of student class schedules, work commitments, and the sometimes wide geographic separation of the students, the prospect of having to work together is at first often resisted. But the students are required to read over the material and to be prepared to form working groups a week later. Where practical, students with similar interests or co workers are accommodated. But an effort is made to form teams with a wide range of experience.

All of the students are given computer accounts at the Ada Development Facility, which is collocated with the Acoustics Department of the university. A tree structure for accounts is established with students working together sharing a common "group" library and having access to an Ada library maintained by the instructor for all students. The class library contains a wide range of compilation units which the students may find useful during the early development and testing of Ada components. Included are run-time profilers, instantiated input/output packages, facilities for logging terminal session activity, and example programs for controlling the graphics devices provided at the facility. This system is also used by the instructor to post messages and test scores. This system is intended to serve as the primary means for developing and testing routines early on. And some of the projects make use of the peripherals provided. Table 2 summarizes the major aspects of the various projects and those will be expanded on in what follows.

Table 1. Topics for Advanced Ada Course Projects

DESCRIPTION	SYSTEM	NO. STUDENTS
Digital terrain radar simulation for central Pa. Allegheny ridge system	Mainframe with color pixel addressable display	3 to 4
Elevation contour mapping using Ada GKS (Graphics Kernel System)	Mainframe with graphics	2 to 3
Pa. political boundary plotting using FORTRAN coded plot packages	Mainframe with personal computer for graphics display	2
Terminal support aids for the visually handicapped	Personal computer with a commercial speech synthesis system	2 to 3
High speed access to large data base	Personal computer with commercial WORM device (Write Once, Read Many)	2
Data acquisition and signal processing system	Personal computer with commercial analog I/O system	2 to 3
AI in Ada implementing Earley's algorithm in parallel	Mainframe	2
Signal processing tools using residue arithmetic	Mainframe or personal computer	2

Once the students have formed into teams, each team is required to present its understanding of the project to the rest of the class. That serves to refine the definition of what is to be done since the class members serve as the customer for the software. It is intended that these presentations serve as a forum for discussion and as a means for everyone to begin to understand what the project is to accomplish. This is also a time for the team to present the breakdown of efforts and schedule for the completion of components parts.

Most of the projects also require access to other systems and this is handled by using personal computers for interfacing. These PCs serve to transfer files between the Ada Development Facility and an IBM mainframe running IBM Ada under CMS. The PC is also used with this mainframe for graphics output. Or, the PCs are used directly to control a speech synthesizer, laser disk record system or analog to digital converter. Terminal emulation software is provided and those students who use the IBM mainframe are granted access to data files prepared by the instructor.

Each team gives a status report about three weeks after the initial presentation. One of the principal items stressed at that time is what, if any, simplifications the effort requires for successful completion by the end of the course. By this time the group members have probably encountered peculiarities which have impacted the way the project is implemented.

Near the end of the course, the teams present a structured walk-through of the code to accomplish the chosen task. Successful completion of the project plays a large part in the final grade. And each student's contribution is evaluated along with the overall success of the team to accomplish what it set out to do. Students each prepare a summary report of that portion of the effort they accomplished, and the team as a whole is responsible for preparing a final written report detailing the work. The instructor participates in a demonstration of the programmed effort for each team. It is not practical, however, for all of the class members to witness each effort's product. So the final walk-through also serves as a forum for describing user interfaces, means for testing the software, and expected behavior when exceptional conditions are encountered.

Table 2. Principle Features of the Projects

PROJECT	FEATURE
Digital terrain radar simulation	Low-level control of color graphics device High efficiency search strategy for large data base Efficient reflectivitely, shadowing computations Minimal display updating using algorithm to determine changes
Elevation contour mapping	Efficient data base search to trace closed boundaries GKS interfaces for polyline plotting and labeling Interactive setup for user selection
Political boundary plotting	Inter-language interfaces with commercial plotting product Operator interfaces for setup to include text pattern matching Interfacing with personal computer for graphics display
Aids for visually handicapped	Low-level setup for speech synthesis system (i.e. mode control) Reactive keyboard input and background output Efficient dictionary search for improved intelligibility
High speed data base access	Low-level initialization and control of SCSI device interface Use of Calendar package for timing and file selection processing Optimized use of defect mapping for expeditious data transfer
Data acquisition and signal processing	Low-level initialization and control of device Pixel mapped graphics for display of data Optimized data processing algorithms
AI in Ada, Earley's algorithm	Definition of algorithm in parallel Arrays of task objects, recursion and generic facilities
Residue arithmetic	Mapping of residue arithmetic types to Ada constructs Generic numeric packages with user defined operators Use of run-time profiling or Calendar package for timing comparisons

An Overview of the Projects

The most ambitious project consists of a simulation of digital terrain radar mounted on a low speed aircraft and flying over the ridge system of central Pennsylvania (Figure 1). The data base consists of in excess of twenty million elevation points organized in South-North slices. One of the major difficulties is locating a specific terrain entity which is swept by the radar beam. So a first step in processing the data is to fly the aircraft at right angles (E/W) to the elevation stripe. And at the range where the radar reflection occurs, one can treat the points in one stripe as being illuminated at the same time. Given the down-range and cross-range elevation data, it is quite simple to compute the reflectivity from the local slope and radar beam angle from the horizontal. Note that this is a significant simplification because true radar reflectivity depends on the ground cover as well as the slope. But the data base does not include information about the type of ground cover.

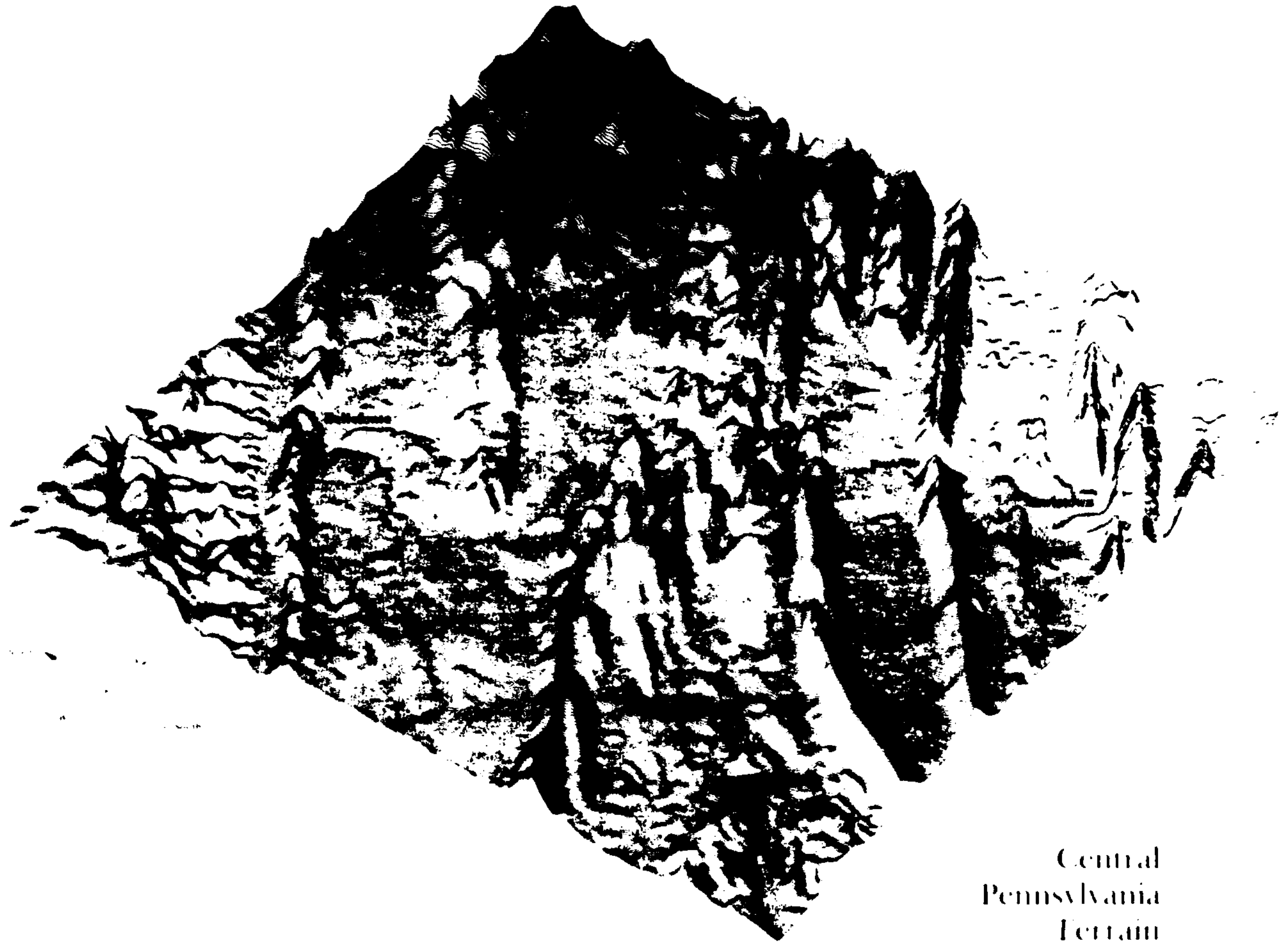


Figure 1. Perspective Plot of the Ridge System of Central Pennsylvania. This is part of the data base used for two of the student projects.

Another difficulty is that of shadowing. High ridges between the aircraft and the terrain slice being displayed can lead to regions which are in a radar shadow. The reflected energy determines the color of a pixel on a color display. In practice, the speed of the aircraft is limited by the rate at which the display can be updated. And so it is important that only those pixels which change color are updated. Even so, real-time simulation is difficult unless the program is highly optimized and the multi-user computer is lightly loaded.

The second project uses the same data base to plot contour lines. The elevation of the desired contour line is input as is the region to be analyzed. A Graphics Kernel System programmed in Ada is used for plotting the contour lines. This GKS is an implementation of routines from the Ada repository for a specific graphics terminal and for a flat-bed plotter. The GKS is provided to the students as a library of Ada entities. The contour plotting is more difficult than it seems. One must find those regions which are at the same elevation along one contour line. In general many closed curves are needed to define the elevation contours within a display region. The students need to use the facilities of the GKS to draw the contours and to label the result. When more students form the group working on this project, they are also expected to support zoom and pan functions.

In another project, the team must process a large data base which consists of the coordinates which define the outline of political regions in Pennsylvania. The boundary is to be plotted using inter-language interfacing to FORTRAN coded graphics routines maintained by the University Computation Center. But the principal challenge is to accept input which may be misspelled or otherwise incomplete and to select plausible alternatives from which the user can choose. Also, the project points out the unique problems posed by Ada ASCII character representation in a machine which inherently handles characters represented in EBCDIC. This mismatch in the language and the environment surfaces as soon as the Ada program attempts to ship the escape sequence to shift the terminal emulation software into graphics mode. An ASCII '[' simply cannot be output from the Ada program!

The fourth project uses a personal computer to which is interfaced a commercial speech synthesis system. A PC based Ada compiler is used for this and other projects which need to be run on personal computers. The software, which has been tested with a mainframe compiler, is down-loaded and low-level interfaces are added to control the hardware. To be useful for the visually handicapped, the speech synthesis system needs to be able to echo character for character what is typed on the keyboard. But information received via serial ports, or output by a running program is enunciated as words. The synthesizer uses hardware to perform the text to speech conversion but it does so in a limited way. Often words need to be purposefully misspelled to be pronounced intelligibly. So the project needs to include a high speed dictionary search which adds features to improve the quality of the conversion to speech. A binary search through a dynamically varying list is one possibility. The dictionary needs to be easily revised by the user to add new words as they occur. Also, low-level features need to be added to allow user keyboard input on a priority basis. The PC based compiler has packages for interfacing with the operating system and those can be used to advantage by the students.

Another personal computer based project makes use of a Laser Disk system (WORM, Write Once, Read Many). The low level interfaces are used to control a resident driver which handles reading. The diskette sized disk stores over 100 million bytes and high speed data transfer requires careful attention to the unique features of this sort of device (Figure 2). As many as fifteen percent of the sectors can be bad and these are marked as such when the disk is formatted prior to sale. When reading, only contiguous good sectors can be processed in one read request. Efficient reading requires that a map of unusable sectors be created and referenced when accessing data in the files. With care, it is possible to achieve data transfers which in most cases waste no disk revolutions.

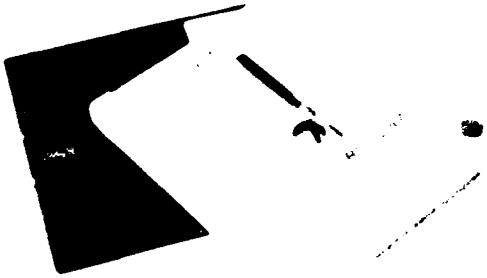


Figure 2 WORM (Write Once, Read Many) Laser Disk Data Cartridge. Shown with a five inch diskette for size comparison

The inputting of digitized signal data via an analog to digital converter using a PC based conversion system is another application which requires low-level control of hardware. In this case the Ada programs need to be interfaced with assembly code to allow access to processor ports. Another aspect of this project is the mapping of data to graphics pixels for the purpose of displaying portions of the input data. Signal processing algorithms need to be developed to do spectral analysis and the spectra are also subject to plotting. The type of signal processing which is supported depends on the number of students working together. And they can choose to implement analysis or digital filtering programs. Good software engineering practice requires localizing all of the low level features in concisely written and documented packages.

The last two projects are more experimental in nature and are suggested to those students who wish to attempt Ada developments which have not previously been reported. The first of these consists of an implementation of Earley's algorithm for string parsing (with AI application) using parallel processing (Reeker, Kreuter and Wauchope, 1985). In this effort, the rewrite rules are maintained in matrix form and the string to be parsed is processed in parallel one symbol at a time and the parse proceeds by recursive application of these rules. The programming effort will result in generic packages which include arrays of tasks to handle the concurrency. A challenge to the students is to present the effort in an understandable way and to develop tests which will demonstrate the parse algorithms for representative AI constructs.

An even more problematic undertaking is the implementation of signal processing routines using residue arithmetic. Ada provides numeric types and operations on those types (fixed point numbers) which are potentially usable with residue arithmetic techniques (Szobo and Tanaka, 1967). This project is not to build a simulation of a residue arithmetic computer, but to develop algorithms which in fact exploit the speed potential of this mathematical approach. For example, a digital correlator using residue techniques may be implemented using table lookup. The students who work on this project need to be well versed in numeric techniques.

Conclusions

The range of projects which we use in the advanced Ada course is designed to interest a large range of students while at the same time requiring development of demanding software applications. And because of the variety of experiences the students encounter, and share with their classmates, they gain an appreciation of Ada in various realistic settings. Sometimes they find that Ada does not live up to expectations because of differences imposed by the machine environment. They also come to appreciate that Ada can be used in applications which have traditionally been done in machine languages. At least, the students learn about working together in an Ada environment to achieve a difficult goal.

References

Reeker, L. H., J. Kreuter and K. Wauchope, "Artificial Intelligence in Ada: Pattern-Directed Processing," Tulane University Report 85-12, May 1985.

Szabo, N. S. and T. I. Tanaka, "Residue Arithmetic and its Applications to Computer Technology," McGraw Hill, 1967.

Claus P. Janota, Ph.D.
Applied Research Laboratory
The Pennsylvania State University
P. O. Box 30
State College, PA 16804

Group leader for systems analysis in the Systems Engineering and Acoustics Department of the Applied Research Laboratory of The Pennsylvania State University. Also, member of the graduate faculty in Acoustics and instructor in the Continuing Education Program. Over fifteen years experience in systems analysis and simulation. Coded the control portion of a six degree of freedom model for an underwater vehicle. The model uses concurrency to decouple the autopilot subsystem from the vehicle response modeling and data collection functions.

QUEUEING NETWORK MODELING AND SIMULATION

Orkun Hasekioğlu

General Electric Company

Abstract

Queueing networks have been extensively used in modeling and analysing communication networks and operating systems. QNETSIM is an object oriented queueing network simulation tool. In QNETSIM queueing networks are composed of five primitive components: generators, absorbers, servers, routers and queues. The following performance measures of an arbitrary queueing network can be simulated: queue length statistics; packet delay statistics; utilisation of servers, routers and absorbers; network throughput. Examples from the comunications network problems include a ring network, static routing and isarithmic flow control.

1 Introduction

Simulation is the ultimate tool for developing and analyzing expensive and large dedicated systems that are difficult to modify for experimental purposes. A communication network is a typical example of this. In many cases, it is desirable to predict the system performance and needed to optimize various design parameters before constructing the system.

Queueing networks have been extensively used to model and analyze the performance of different multiprogrammed computer systems, operating systems and communication systems [1,2,3].

In the sequel, a queueing network simulation approach using process interaction view of simulation is described. The modeling applications of queueing networks are so widespread that QNETSIM can be used for many other purposes, including manufacturing system assembly line simulation.

Following the general description of QNETSIM, the basic components of QNETSIM are introduced. The applications are demonstrated by several examples, including a ring network, flow control and routing examples.

2 General Description of QNETSIM

QNETSIM is an Ada based queueing network simulation package, so that powerful programming language capabilities of Ada can be utilized.

In QNETSIM a queueing network is considered to be composed of five components: generator, absorber, queue, server and router (Figure 1). The detailed functionalities of these components are described later. The generators, absorbers, servers and routers are treated as active processes in the process oriented simulation approach. On the other hand the queues are passive consructs. Packets or jobs are stored in the queues. Deleted from the queues and processed by the active components.

The specific parameters of these components are defined through their attribute tables (Figure 2). These tables contain all the necessary information to construct any queueing network using the five basic queueing network components mentioned above. The packets (jobs) processed inside the network are also defined by certain attributes. Although these attributes can be expanded based on other needs, it is considered to be appropriate to include the following attributes:

- Packet length is the processing time for a particular job. In communication network simulation packet length usually denotes the tranmission time of a packet. This is determined by the generator process, based on sampling a predetermined statistical distribution.

- Packet priority field is utilized to process packets in a priority order, if necessary.

- Total packet delay field is updated each time a packet encounters a delay and it is used to generate statistical performance measures on packet delay.

- Packet serial number: each packet departing from a generator is assigned a serial number that my be utilized to trace a particular job through the network.

- Packet type: In communication systems simulation, for example, generated packets have certain types, as required by the communication protocol, e. g. , acknowledgement, information etc. .

The input to QNETSIM can be either interactive, or through a configuration file that contains the structural information on the queueing network. Similarly, statistical results of the simulation are either directed to the terminal or to a data file that can further be used for evaluation or animation processes.

3 Performance Measures Considered

QNETSIM produces statistical results on all major widely accepted performance criteria in queueing network analysis. These include the statistics on queue (buffer) sizes, utilization (busy probability) of packet processors, i. e. , absorbers, servers and routers, statistics on packet delay at any absorber, input/output rates of packet arrivals and departures to and from the networks. The performance results also include the throughput of the network, defined as the ratio of the number of packets leaving the network to the number of packets entering the network.

4 Queueing Network Modeling Components

Queueing network models consist of five primitive components: generator, absorber, queue, server, router. Simulation models should be composed of these five elements. Here, we describe the functionalities of these components or processes.

4.1 Generator

Generators are the external sources that generate packets to be transmitted through the network. Based on the statistical parameters specified in the generator attributes table, it produces packets. For identification purposes, each generator is assigned a number. The attributes of a generator are the following:

1. Packet interarrival distribution type and the statistical parameters of the distribution. The interarrival time distribution may be any one of exponential, gaussian, erlang, uniform or constant. The statistical parameters related to the packet interarrival distribution have to be specified, e. g. , mean, variance, erlang parameter etc.
2. Packet length distribution type and the statistical parameters of the distribution. Packet length distribution types and the related statistical parameters are the same as the ones for the packet interarrival distribution.
3. Priority. As the packets are generated they are assigned a specific priority by the generator. The priority of the packets can be utilized to implement different scheduling procedures for the processing of packets.

The identifying number of the queue that the generator inserts the packets is to be specified as well. Generators also keep track of the total number of packets generated during the simulation time, in order to be able to calculate some of the performance measures. Any number of generators can be connected to a queue. In a physical situation, depending on the application, a generator may represent a transmitter.

4.2 Queue

Queues are the entities where the packets are stored and then processed, based on the queueing discipline specified. The generators connected to the queue, insert the packets to the queue. Each queue is assigned a number for identification. The attributes of a queue are the following:

1. Queue length is the maximum number of packets that can be stored in the queue simultaneously. If not specified, it is considered to be infinite.
2. Queueing discipline, is the scheduling discipline that the stored packets are processed. It can be first-come-first serve (FIFO), last-in-first-out (LIFO) or priority where the packets are processed in the order of their priorities. High priority packets are processed first.

In a practical situation, a queue may be used to model a buffer or an abstract feature of a communications protocol.

4.3 Absorber

Absorbers are the entities where the packets exit the queueing network. Absorbers connected to the queues delete the packets from the queue and process the packets based on the information specified in the attribute fields of the packets. Each absorber has an identification number and a specified queue number that the packets are deleted from. In a practical case, an absorber may represent a receiver that deletes the packets from the receiver buffer.

4.4 Server

Servers delete the packets from the queue they are connected to, delay for the period of time specified in the packet length attribute field and insert the packet again to a specified queue. Therefore, the identification numbers of the queues from which the server deletes the packets and to which the server inserts the packets have to be specified. In a communication network simulation model, servers may represent the delay caused by the transmission time of a packet.

4.5 Router

Basically routers have the same functionalities as the servers, except that it can insert the packets to more than one queue based on a routing policy associated with it. The routing policy specifies the probability of branching to a particular queue.

Examples are given to illustrate the way these simulation model building components are utilized in real world situations.

5 Examples

1. A ring network:

   Each node consists of a packet generator (transmitter), an absorber (receiver), a router, receive and transmit buffers. Packets are circulated around the ring network. With a certain probability the packet is retained in a node and with another probability it is transmitted to the next node. Sample input configuration file and output statistical results files are given (Figure 4,5).

   Figure 3 illustrates a 4-node ring network. Each of the absorbers, generators, routers and queues are identified by an integer identification number. The two queues in each of the nodes represent the input and output buffers. Absorbers correspond to receivers and generators correspond to transmitters. Figure 4 is a sample input file for a particular set of parameters. For each transmitter (generator), the corresponding transmitting buffer, statistical information on the packet lengths and interarrival time and their identification numbers are supplied. The required information for the receivers (absorbers) are their identification numbers and the asociated receiving buffers. As part of the link information the branching probabilities are supplied. Finally, the simulation length in the units of time and the number of independent trials of the simulation experiment are given. The simulation program is repeated each time with the same parameters with new random seeds for the random number generators.

The output statistical results (Figure 5) include the statistics on the buffer sizes, utilization of the transmitters, receivers and links, statistics on packet delay, the input and output rates of packets and the throughput. Throughput is defined as the percentage of the packets that exit the network. These are the most commonly referred performance parameters in the analysis of queueing systems. We note that the system is still in the transient phase of the end of 100.0 simulation time units. Many of the packets have not reached the receiving buffers, yet. We can observe this by comparing the input and output rates, and from the queue length statistics.

2. Routing:

   A static routing procedure, where every node has a constant routing table indicating the manner the incoming traffic is distributed to the adjacent nodes can be identified by the routing policy and a traffic distribution matrix. Each element, (i, j), of the NXN traffic distribution matrix, where N is the number of nodes, indicates the external traffic arrival rate from node i to node j. The routing policy attributed to each router element specifies the probabilistic distribution of the packets to the adjacent nodes. The delay and queue sizes can be optimized by varying the routing procedure until the optimum desired results are achieved. In this example (Figure 6), the nodes have the identical queueing structure as in the ring network example.

3. Isarithmic Flow Control: In isarithmic flow control (Figure 7), an external packet arriving at a node of the network is permitted to enter the network, if the node has a permit [3]. If not, the packet must wait until the node gets a permit packet from another node. This way the maximum total number of transmitted packets waiting in the network is limited by the total number of permit packets in the network. This situation is usually modeled as a double ended queue typically observed at a taxi stand. Therefore, transmission occurs only if there is a permit packet in the permit queue and in the external packet queue simultaneously. If an external packet can not find any permit packet in the permit queue, it delays its transmission until a permit packet arrives.

6 Overview and Future Improvements

QNETSIM is developed as a multipurpose queueing network simulation package base on Ada and process oriented simulation techniques. It is capable of simulating many scenarios encountered in the modeling of communication systems where queueing networks are utilized. Nevertheless, some of the communication protocol modeling problems are not directly representable in the form of queueing networks. In such cases, the resulting queueing models may represent oversimplified versions of the original protocol, so that reliable simulation results are difficult to generate. To overcome such problems a new approach is being developed where queueing network simulation is incorparated with formal protocol representation techniques, such as, formal programming language description and finite state machine representations of communication protocols.

References

[1] C. H. Sauer and E. A. MacNair, *Simulation of Computer Communication Systems*. Prentice-Hall, 1983.

[2] M. Schwartz, *Telecommunication Networks, Protocols, Modeling and Analysis*. Addison-Wesley, 1987.

[3] H. Inose, *Digital Integrated Communications Systems*. University of Tokyo Press, 1979.

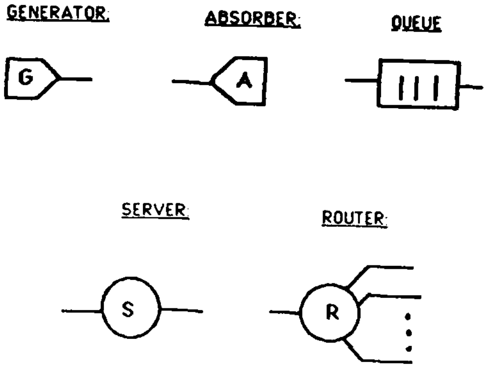


Figure 1: Queueing networks are composed of the five primitive elements.

QUEUE ATTRIBUTES :

Queue number
Queue Length
Queueing Discipline (FCFS, LIFO etc.)

GENERATOR ATTRIBUTES:

Generator number
Inter-arrival Distribution
Average inter-arrival time
Packet Length Distribution
Average Packet Length
Priority
Number of the queue to insert the packets
Total number of packets generated

ROUTER ATTRIBUTES:

Router number
Total number of Branches
Probability of Routing to each Branch
The number of the Queue each Branch is connected to
The number of the queue the Router serves

SERVER ATTRIBUTES:

Server number
The number of the queue served
The number of the queue the packets are inserted to

ABSORBER ATRRIBUTES:

Absorber number
The number of the queue served
Total number of exiting packets

PACKET ATTRIBUTES:

Packet Length
Packet Priority
Total Packet Delay
Packet serial number
Packet Type

Figure 2: Attributes of the entities in a queueing network.

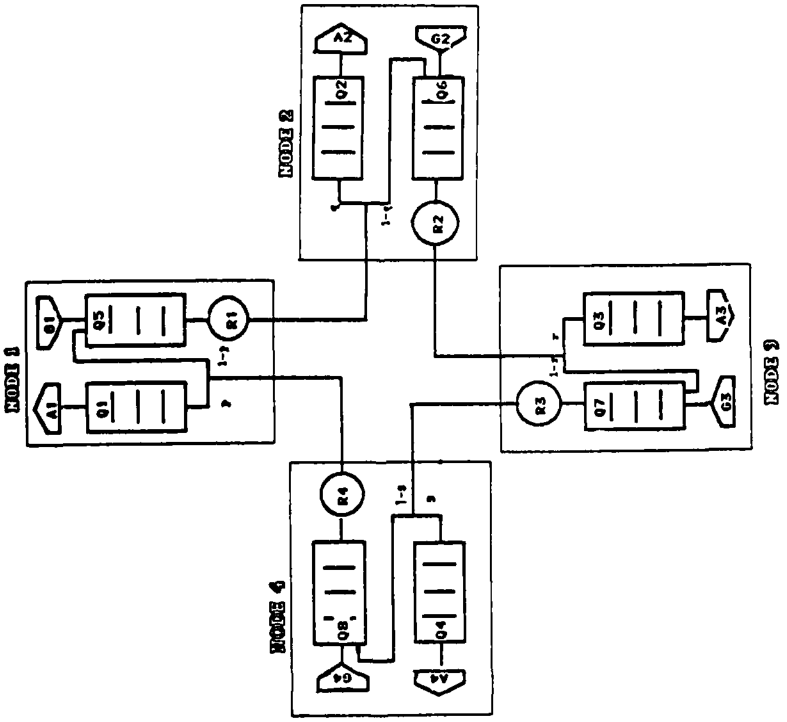


Figure 3: A 4-node ring network.

```
NUMBER OF BUFFERS                  8

NUMBER OF TRANSMITTERS:            4
    TRANSMITTER 1
    TRANSMIT BUFFER                     5
    PACKET LENGTH DISTR                 exponential
    INTERARRIVAL DISTR                  exponential
    AVG. PACKET LENGTH                  0.5
    AVG. INTERARRIVAL                   1.0
    TRANSMIT TO                         1
    TRANSMITTER 2
    TRANSMIT BUFFER                     6
    PACKET LENGTH DISTR                 exponential
    INTERARRIVAL DISTR                  exponential
    AVG. PACKET LENGTH                  0.4
    AVG. INTERARRIVAL                   0.9
    TRANSMIT TO                         2
    TRANSMITTER 3
    TRANSMIT BUFFER                     7
    PACKET LENGTH DISTR                 exponential
    INTERARRIVAL DISTR                  exponential
    AVG. PACKET LENGTH                  0.3
    AVG. INTERARRIVAL                   0.6
    TRANSMIT TO                         3
    TRANSMITTER 4
    TRANSMIT BUFFER                     8
    PACKET LENGTH DISTR                 exponential
    INTERARRIVAL DISTR                  exponential
    AVG. PACKET LENGTH                  0.6
    AVG. INTERARRIVAL                   0.8
    TRANSMIT TO                         4

NUMBER OF SERVERS                   0

NUMBER OF RECEIVERS                 4

        RECEIVER 1
         RECEIVE BUFFER                 1
        RECEIVER 2
         RECEIVE BUFFER                 2
        RECEIVER 3
         RECEIVE BUFFER                 3
        RECEIVER 4
         RECEIVE BUFFER                 4

NUMBER OF LINKS                      4

        LINK 1
          TRANSMIT BUFFER               5
          NUMBER OF BRANCHES            2
             BRANCH 1
                BRANCH PR.                  0.2
                BUFFER #                    2
             BRANCH 2
                BRANCH PR.                  0.8
                BUFFER #                    6

       LINK 3
         TRANSMIT BUFFER                7
         NUMBER OF BRANCHES             2
            BRANCH 1
               BRANCH PR.                   0.8
               BUFFER #                     4
            BRANCH 2
               BRANCH PR.                   0.2
               BUFFER #                     8

       LINK 4
         TRANSMIT BUFFER                 8
         NUMBER OF BRANCHES              2
            BRANCH 1
               BRANCH PR.                    0.6
               BUFFER #                      1
            BRANCH 2
               BRANCH PR.                    0.4
               BUFFER #                      5

 SIMULATION TIME                       100.0
 NUMBER OF TRIALS                      1
```

Figure 4 : A sample input network configuration file for the 4-node ring network.

```
                STATISTICAL RESULTS

The AVERAGE LENGTH of BUFFER        1 is  1.94214E-03
The MAX LENGTH of BUFFER            1 is            1
The MIN LENGTH of BUFFER            1 is            0

The AVERAGE LENGTH of BUFFER        2 is  1.48981E-02
The MAX LENGTH of BUFFER            2 is            1
The MIN LENGTH of BUFFER            2 is            0

The AVERAGE LENGTH of BUFFER        3 is  2.91321E-03
The MAX LENGTH of BUFFER            3 is            1
The MIN LENGTH of BUFFER            3 is            0

The AVERAGE LENGTH of BUFFER        4 is  5.90820E-04
The MAX LENGTH of BUFFER            4 is            1
The MIN LENGTH of BUFFER            4 is            0

The AVERAGE LENGTH of BUFFER        5 is  1.41775E-01
The MAX LENGTH of BUFFER            5 is            4
The MIN LENGTH of BUFFER            5 is            0

The AVERAGE LENGTH of BUFFER        6 is  3.75138E-01
The MAX LENGTH of BUFFER            6 is            4
The MIN LENGTH of BUFFER            6 is            0

The AVERAGE LENGTH of BUFFER        7 is  3.44925E+00
The MAX LENGTH of BUFFER            7 is            8
The MIN LENGTH of BUFFER            7 is            0

The AVERAGE LENGTH of BUFFER        8 is  3.20127E-01
The MAX LENGTH of BUFFER            8 is            4
The MIN LENGTH of BUFFER            8 is            0


The utilization of the ABSORBER          1 is  1.94214E-03

The utilization of the ABSORBER          2 is  1.24152E-02

The utilization of the ABSORBER          3 is  3.14270E-03

The utilization of the ABSORBER          4 is  5.90820E-04

The utilization of  LINK         1 is  3.69175E-01

The utilization of  LINK         2 is  4.83719E-01

The utilization of  LINK         3 is  8.13851E-01

The utilization of  LINK         4 is  4.74684E-01
```

```
    ABSORBER            2 :

    The AVERAGE DELAY of the packets leaving the network is 9.12426E-02
    The MAX DELAY of the packets leaving the network is 2.73728E-01
    The MIN DELAY of the packets leaving the network is 1.56432E-02

    ABSORBER            3 :

    The AVERAGE DELAY of the packets leaving the network is 5.94474E-02
    The MAX DELAY of the packets leaving the network is 1.78342E-01
    The MIN DELAY of the packets leaving the network is 1.78453E-02
The output rate through the ABSORBER    1 is  3.00000E-02 packets/unit time

The output rate through the ABSORBER    2 is  3.00000E-02 packets/unit time

The output rate through the ABSORBER    3 is  3.00000E-02 packets/unit time

The output rate through the ABSORBER    4 is  3.00000E-02 packets/unit time

The input rate through the GENERATOR    1 is  1.02000E+00 packets/unit time

The input rate through the GENERATOR    2 is  1.14000E+00 packets/unit time

The input rate through the GENERATOR    3 is  1.87000E+00 packets/unit time

The input rate through the GENERATOR    4 is  1.27000E+00 packets/unit time

The THROUGHPUT of the network is 3.20000E-02
```

Figure 5 : Statistical results for the configuration file given in Figure 4.

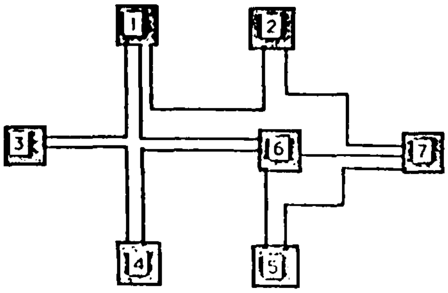


Figure 6: Graphical simulation model for the static routing example in a communication network.

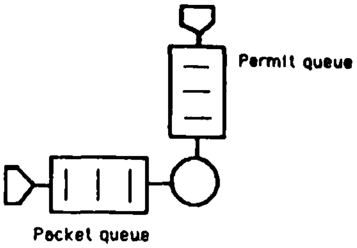
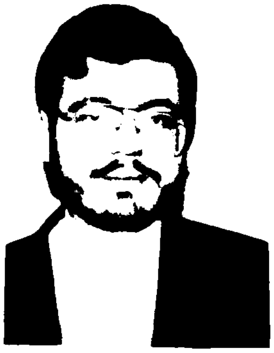


Figure 7: Queueing model for isarithmic flow control in communication networks.



Orkun Hasekioğlu was born in Ankara, Turkey, in 1962. He received the B. S. degree in Electrical Engineering from Polytechnic Institute of New York, in 1983, and the M. S. degree in Electrical Engineering from California Institute of Technology, in 1984. He is expected to receive Ph. D. in Electrical Engineering from Rensselaer Polytechnic Institute by May 1988. Currently, he works as a research and consulting engineer with General Electric Company, Corporate Research and Development, Schenectady. His current research interests are performance modeling and analysis of communication networks.

Address: Orkun Hasekioğlu, P. O. Box 1135, Troy, N. Y. 12181.

The Suitability of Ada for Communications Protocols

Robert H. Pollack and David J. Campbell

Unisys Defense Systems
System Development Group
Paoli Research Center

Abstract

Communications protocol implementations are a class of software with unusual requirements. These programs must not only perform well in real time, but must also satisfy rigid formal specifications. In the past, protocols have chiefly been implemented in C or assembly language. In this paper, the authors present the results of a study of protocol implementations in Ada*. The influence of several Ada language features on protocol performance is measured in two ways, first by "standard" benchmarks, and second by varying the use of these features in an Ada protocol implementation. Finally, the authors compare the performance of Ada and C implementations of the same protocols, concluding that the protocol community can take advantage of Ada's modularity and abstraction mechanisms without sacrificing performance.

1. Communications Protocol Software

The implementation of communications protocol software has requirements that are not shared by the bulk of data-processing or scientific programs. These implementations differ from most other software in requiring both conformance to standards and in performance constraints.

In the first place, protocol software must often conform to rigid formal specifications. This is particularly true of implementations of standard protocols, such as those specified by the Department of Defense or the International Standards Organization. It is only complete conformance to standards that makes it possible for different implementations to communicate at all.

It is worth pointing out that conformance to protocol standards differs significantly from the conformance to design that is required by all software implementations. Unlike most software designs, a protocol standard will seldom change during the implementation phase; even if design changes occur, they will never be under the control of the implementor. In addition, the designer and the implementor will seldom be the same person or even communicate with one another. A protocol standard is a software design which must speak entirely for itself.

The second requirement of protocol implementations is that they perform well in real time. This requirement arises from two considerations. First, most protocol implementations are "utility" software, i.e., software that is run repeatedly or continuously, as opposed to programs that are run infrequently. Second, individual protocol standards may have specific real-time requirements, e.g., they may require that a network node respond to a certain request within a given number of seconds or even within milliseconds. This second requirement of protocol software is shared by embedded systems; indeed, many protocol implementations are themselves embedded systems or are part of embedded systems.

These two requirements of protocol software are often perceived to be in conflict, particularly in determining a suitable implementation language. On one hand, conforming to abstract formal standards suggests an implementation language that permits a high degree of abstraction, and one which contains mechanisms for insuring that these abstractions are used consistently. On the other hand, the requirement for effective real-time performance suggests a language that is "close to the machine."

This paper reports on the effectiveness of Ada in meeting both of these goals.

2. Overall Approach

In the past, implementation clarity and conformance to standards have usually taken second place to performance requirements. Hence, most protocol implementations have been in assembly languages or in C. While the use of such languages have provided the requisite performance, the resulting implementations leave much to be desired in the areas of portability, maintainability, and interoperability.

It was exactly these difficulties in embedded systems that triggered the development of Ada [Barnes84]. Moreover, Ada offers abstraction mechanisms that closely correspond to those used in protocol specifications (see, for example, [Castanet86]). The most significant remaining question, therefore, is whether existing Ada compilers can generate sufficiently good code to meet protocol

* Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

The work described herein was performed under Defense Communications Agency contract number DCA-100-83-C-0064.

performance requirements. The work described here shows the results of a study commissioned by the Defense Communications Engineering Center (DCEC) of the United States Defense Communications Agency (DCA) to investigate the performance of protocol implementations written in Ada.

Our overall approach considered the problem on several independent axes. The first was a comparison of three different Ada compilation systems with respect to the performance of those Ada features which both DCEC and the authors regarded as important in the protocol domain. Second, we attempted to determine the influence of various Ada features on the performance of an Ada implementation of the DoD protocols TCP and IP. Finally, we compared the performance of this Ada implementation to one written in C.

The test environment consisted of two VAX[†] 11/785s, one under Berkeley UNIX[‡] and one under VMS. The compilation systems studied were

- the Verdix Ada Development System (VADS) [Verdix86], version 5.41(e), under UNIX 4.3 BSD
- Digital Equipment Corporation's VAX Ada (DEC Ada) [DEC85], version V1.3-24, under VMS V4.5
- the Ada Language System/Navy (ALS/N ADAVAX) [ALSN87], version 3.99, under VMS V4.5

3. Compiler Benchmarks

Each of the compilers described above was benchmarked using a specially tailored benchmark suite, designed to test Ada features important in protocol implementations. The purpose of this benchmark suite was to permit the performance of each compilation system to be analyzed at the individual feature level.

3.1. Benchmark Suite Design

Based on the special requirements of protocol software, the authors developed a set of specific criteria by which Ada compilation systems would be evaluated. These criteria included measurements of execution speed in the areas of tasking, exceptions, and the speed of basic constructs, such as procedure calls, storage allocation, array and record component references, and simple integer arithmetic operations. The suite omitted measurements of features not frequently used by communications protocols, such as floating point arithmetic.

The benchmarks were taken from three public-domain benchmark suites:

- the Prototype Ada Compiler Evaluation Capability (ACEC) [Hook85]
- the PIWG benchmark suite, developed by the Performance Issues Working Group of the ACM [Squire85]
- the Hughes Aircraft Company Ada Benchmark Suite [Hughes86].

3.2. Benchmark Results

The benchmarks were run on two VAX 11/785s, one running 4.3 BSD UNIX, and the other VAX VMS. Each system was free of other programs during the benchmark runs. The results given below are based on the best performance of at least three samples of each test. Each test was compiled with all Ada run-time checks suppressed and with full optimization.

To aid in comparison, we express most of our results as ratios. In each test, we arbitrarily define the performance of the Verdix compiler as 1.0, and express the other measurements as multiples of this measurement, i.e., we *normalize* all measurements with respect to VADS. The results given in this section are all normalized except where otherwise indicated. Moreover, the normalized results in this section are the ratios of *time*; that is, a normalized result of, say, 2.0 for a given test means that that test took twice as long to run as the same test under VADS. The reader should also be aware that all of our measurements, whether normalized or not, are measurements of *execution* time only, not compilation time.

Although we collected both CPU time and elapsed time for all tests, the measurements below are based on CPU time. We found that for nearly all our samples, these two measurements were extremely close (within 1% of each other). Elapsed time, however, was occasionally larger than CPU time. Since this effect appears to be random, we believe that it is due to operating system interference, and therefore regard CPU time as the more reliable measurement.

Tasking

Several different benchmarks were used to evaluate the run-time systems with respect to tasking. These included the Hughes tasking paradigm benchmarks and a group of ACEC benchmarks in the tasking area. Figure 1 summarizes the results of these benchmarks.

Benchmark	Compiler		
	ALS/N ADAVAX	DEC Ada	VADS
Hughes	5.7	2.1	1.0
ACEC Run-Time	4.3	2.0	1.0

Figure 1. Tasking Benchmarks (normalized)

The first row of this table shows the mean result of five tests from the Hughes benchmark suite which measure the performance of various interactions between producer and consumer tasks, ranging from a simple interaction to more

[†] VAX is a trademark of Digital Equipment Corporation

complex inter-task communication involving buffering and transporter tasks. The second row shows the mean result of ten ACEC tests which measure the efficiency of the run-time system's task handling. This group includes the ACEC task chaining tests and the select and guard tests.

The authors also developed their own test to measure the number of small tasks that each run-time system could support. This test consisted of simply starting as many small tasks as possible. We found that all three systems reach a point at which they no longer will initiate new tasks, but that no system raises an Ada exception when this point is reached. Note that the ALS/N ADAVAX was able to initiate more than twice as many tasks as either of the other systems. Figure 2 shows the observed task capacity of each compilation system, to the nearest 100 tasks.

Benchmark	Compiler		
	ALS/N ADAVAX	DEC Ada	VADS
Tasks	1900	800	900

Figure 2. Task Capacity (number of tasks)

Exception Processing

The time to raise and propagate an exception in various situations was measured using three PIWG exception-handling tests. Figure 3 shows the mean of the normalized results of these tests.

Benchmark	Compiler		
	ALS/N ADAVAX	DEC Ada	VADS
PIWG Exceptions	0.05	0.07	1.00

Figure 3. PIWG Exception Handling Tests (normalized)

General Execution Speed

The execution efficiency of basic constructs was measured by a large set of benchmark programs taken from the ACEC benchmark suite. This set consists of the tests in the ACEC suite which are designed to measure only the speed of basic constructs, common to all programming languages. The ACEC tests which measure the performance of constructs found only in Ada, such as tasking and exceptions, are discussed elsewhere in this paper. In addition to this set, we used four PIWG benchmarks, described below. We also measured execution speed with the well-known Dhrystone test. We discuss each of these results separately below.

The tests chosen from the ACEC suite included measurements of the speed of **for** loops, **case** statements, variable reference, parameter reference, and integer arithmetic. The set of tests taken from this suite consisted of 34 test *pairs*, each pair consisting of a control test and feature test. The final result for each test was considered to be the minimum CPU time achieved in a group of at least three samples. Figure 4 shows the mean of the normalized differences of the 34 test pairs.

It can be seen from this figure that VADS and DEC Ada execute our chosen tests in about the same time, while the ALS/N ADAVAX executes about 15% faster.

Benchmark	Compiler		
	ALS/N ADAVAX	DEC Ada	VADS
ACEC mean	0.85	0.97	1.00

Figure 4. ACEC Execution Tests (normalized)

Figure 5 shows the results of a set of PIWG benchmarks which measure the cost of allocating dynamic-sized structures during elaboration and deallocating them on subprogram exit. Since the significance of these measurements depends on comparing different rows of the table, we present the raw results, in microseconds of CPU time.

Benchmark	Compiler		
	ALS/N ADAVAX	DEC Ada	VADS
D000001	438.3	4.4	11.7
D000002	7775.0	3328.1	2072.9
D000003	449.6	8.1	1006.5
D000004	12612.5	4787.5	3322.9

Figure 5. Dynamic Allocation (μsec)

The results of these tests show that dynamic allocation can have a considerable influence on the performance of actual programs. They show that each of the compilers has a choice of two different allocation strategies. Moreover, in a given situation, the three compilers do not all choose the same allocation strategy. Test D000001 measures the time to allocate a dynamic-sized array of integers. Test D000002 is the same as D000001, but initializes the array with an **others** clause. Tests D000003 and D000004 are the same as D000001 and D000002, respectively, except that the array is a field of a discriminated record. The two allocation strategies are illustrated by tests D000001 and D000003. One of these strategies takes on the order of 10 μsec, while the other takes several thousand μsec. We believe that the first operation consists of simply expanding the program stack, while second involves allocation from a global heap.

‡ UNIX is a trademark of AT&T Bell Laboratories

As the table shows, the DEC compiler chooses stack expansion for both kinds of uninitialized objects, the ALS/N ADAVAX chooses global allocation for both, while the Verdix compiler uses stack expansion for the array and global allocation for the record. Thus, the programmer's choice of data structures can be seen to have a potentially great influence on performance when moving from one Ada compiler to another.

Figure 6 shows the results of the Dhrystone benchmarks, once again normalized to VADS. Dhrystone is a composite benchmark that represents a "typical" program. Dhrystone uses a special mix of statement types and data types, constructed from a statistical analysis of a large number of "real" programs [Weicker84]. The results in Figure 6 were measured with all of Ada's run-time checks enabled; the influence of these checks on performance is discussed below.

Benchmark	Compiler		
	ALS/N ADAVAX	DEC Ada	VADS
Dhrystone	2.0	0.6	1.0

Figure 6. Dhrystone Results (normalized)

Suppress Pragma

In addition to measuring execution speed of the features described above, we also wished to determine the cost of Ada's many run-time checks. The effect of suppressing all run-time checks was measured by running Dhrystone with all checks on and all checks suppressed. The latter effect can be achieved by using the Ada SUPPRESS pragma, but all of the compilation systems also permit this effect to be achieved by a compile-time option. Dhrystone was also used for this test because it reflects the number of checks that would have to be performed during the execution of a typical program. Figure 7 shows the time in milliseconds for a single Dhrystone execution, both with and without checks. The table also shows the benefit of suppressing these checks, expressed as a percentage of the execution time with checks enabled.

Test	Compiler		
	ALS/N ADAVAX	DEC Ada	VADS
checks	1.48	0.45	0.75
no checks	1.25	0.32	0.52
% change	+16%	+29%	+31%

Figure 7. Effect of Suppressing Checks on Dhrystone (msec)

3.2.1. Benchmark Summary

The execution speed of "ordinary" constructs, as measured by the ACEC suite, was best with the the ALS/N ADAVAX compiler; VADS and DEC Ada showed about equal performance. In the tasking area, the results show that the VADS run-time system's support of tasking performs about twice as fast as DEC's and four to six times as fast as that of the ALS/N ADAVAX. The results of our exception-handling tests show that the ALS/N ADAVAX offers the fastest implementation, while VADS is by far the slowest.

The results of the Dhrystone benchmark show that DEC offers the fastest performance, followed by VADS and the ALS/N ADAVAX. This result is somewhat surprising, since the ACEC suite shows these two compilers to be about equal in performance. One possible explanation is the difference in dynamic record allocation strategies between these two compilers, which we have discussed above. It is also possible that some of the features measured by the ACEC tests play a stronger role in programs than others, a fact which is not reflected by the simple mean of all these tests.

It is worth noting that the Dhrystone results provide a fairly good approximation of the results we obtained in our protocol performance measurements, described below. Thus, our protocol programs proved to be "typical", according to the Dhrystone criteria.

4. Ada Protocol Design

One of the fundamental concepts in protocol design is that of *layered network architecture*. In such an architecture, entities at a particular level or layer communicate with one another by using the services provided by the layer directly beneath. These entities, in turn, provide a well-defined set of services to the layer above. The services that must provided by each layer comprise the *reference model* for a particular architecture; two well-known reference models are the ISO Open Systems Interconnection (OSI) Model [ISO83] and the DoD Protocol Reference Model [DCA82].

TCP and IP are protocols at consecutive layers in a layered network architecture, with TCP running at a level corresponding roughly to the ISO transport layer and IP running beneath it at the network level. The two protocols are almost always used together, and, in particular, are used together in the ARPANET. Following the terminology of the IP standard [DoD83c], we refer to the data link protocol beneath IP as the *subnetwork protocol*, or SNP. In this paper, we refer to any protocol running above TCP as the *upper-layer protocol*, or ULP. For more detail on protocol reference models or TCP/IP, the reader should consult any standard work on computer networks; a good comparison of the ISO and DoD reference models may be found in [Tanenbaum81].

The performance measurement task whose results are described here was part of a larger contract, which also included the creation of complete Ada implementations of the DoD Transmission Control Protocol (TCP) [DoD83a] and the Internet Protocol (IP) [DoD83c]. The design and coding of the Ada version of TCP/IP were carried out by Unisys personnel at the SDG West Coast Research Center. We present a brief overview of this design below.

Both the TCP and IP standards are specified in an Ada PDL. Accordingly, one of the principal design considerations was to use these standards as the implementation model, wherever possible. Following the TCP specification, the communicating TCP entities are implemented as finite state machines (FSMs), with each FSM provided with a means of communicating with the ULP and with IP. The data structures and subprograms in this implementation are those used in the military standard specifications.

The design also stresses portability. Portability is achieved in a number of ways. First, all data types are, as much as possible, explicitly defined. That is, the use of implementation-dependent Ada types, such as INTEGER, is avoided; instead, explicit ranges and storage sizes are given for numeric types. Second, each type is made as constrained as possible. For example, internet addresses are described as arrays of eight-bit characters rather than as 32-bit integers, to avoid problems with Ada implementations that do not support 32-bit arithmetic types. Finally, constructs that are non-portable, or areas where performance may be improved by the use of non-portable constructs, are isolated, using the Ada package mechanism.

The design takes good advantage of Ada modularity by isolating each of the two protocols, TCP and IP, to its own set of packages. Each inter-layer interface is defined in its own package, with the implementation details of the interface hidden from the interface user. We found that this feature of the design made the protocol implementation extremely easy to modify, in ways that are described below.

All concurrency required by the implementation is expressed by Ada tasks. This frees the implementation from any dependency on particular operating systems. Communication between the protocols, as well as communication to the upper and lower layers, is accomplished by means of generic queueing routines.

5. Ada Protocol Testing Results

In order to provide performance measurements that were independent of I/O devices and as independent as possible of the operating system, the Ada protocol implementation was modified to provide for single-node operation. Figure 8 shows a rough overview of the structure of the Ada implementation, and indicates the modifications made for performance measurements. In the figure, tasks are represented by circles and some important packages are represented by squares. The figure by no means shows all of the tasks in the implementation, but shows which tasks figure prominently in the processing of both input and output messages, and which tasks are bypassed by the single-node loopback paths. Since the reader should have an understanding of the task structure of the Ada implementation to understand what follows, we will briefly explain the major elements of the figure.

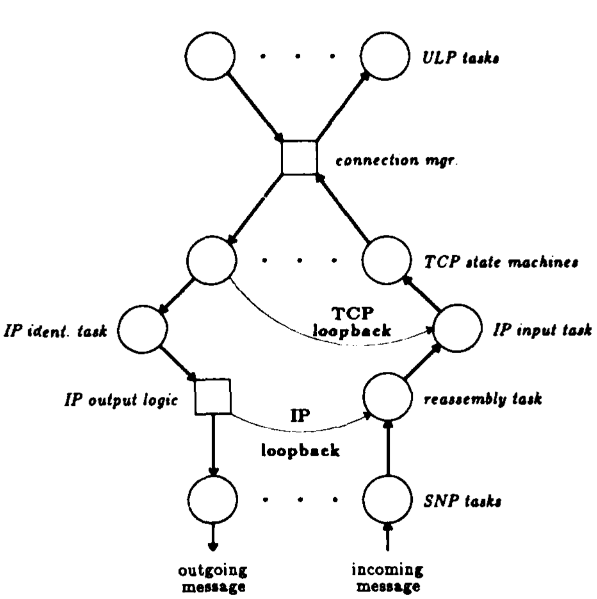


Figure 8. Ada TCP/IP Message Paths

The interface between the ULP and the TCP state machines (FSMs) is through a package known as the *connection manager*, which actually performs the rendezvous with the state machines. The operation of a given FSM is self-contained and the FSM is uniquely associated with a particular ULP task. Thus, no synchronization is required between the connection manager and the FSMs except during the establishment and disestablishment of this association; when needed, this synchronization is provided by another task not shown in the figure.

Outgoing IP messages (known, at this level, as *datagrams*) require a unique identification number. This number is provided by the *IP identification task*. All incoming IP datagrams are processed by a single *reassembly task*, which stores these messages in a single queue for delivery to TCP. The *IP input task* provides synchronization of operations on this queue between IP and the TCP state machines.

In our throughput measurements, only one ULP task was provided, which controlled both the input and output message traffic; we refer to this task as the *driver*. The use of the TCP and IP loopback paths was entirely under the control of the driver, making it possible to measure the performance of each protocol independently of the other.

For the sake of brevity, we limit the measurements discussed in this paper to those which we believe will be of most interest to the protocol community. These are the following:

- The *connect/disconnect time* of TCP state machines. The measurements shown here were obtained through the TCP loopback path, and represent the performance of the bulk of the TCP code.

While most of the TCP code is concerned with establishing and breaking connections, most of the time spent by a TCP/IP implementation is spent transferring data with the connections fully established. Accordingly, the following measurements are also of considerable interest:

- The *data transfer rate* of TCP and IP. These measurements were made through the IP loopback path. They were obtained by exchanging messages between *pairs* of TCP state machines, and so represent the overhead of both sending and receiving messages. In this measurement, the size of each message was the largest that could be sent on a subnetwork whose maximum message size is 1024 characters, a typical value for actual store-and-forward networks. The actual data size of each message was 904 characters, to allow for TCP and IP message headers, and the transfer rates shown are based on this number.
- The *message transfer rate* of TCP and IP. Protocol implementations are often compared by the number of messages they can process per unit time. To facilitate such comparisons, we present measurements of the number of short messages that could be exchanged through the IP loopback path. In addition to the TCP and IP message headers, each message contained only one character of user data.

A number of variations were made to the Ada protocol to establish the effect of Ada features on protocol performance. These variations were in the areas of tasking, timer management, and message formatting. The results of these variations are reported and discussed below.

The measurements presented in this section are based on elapsed time, on systems that were free of other users. Each reported measurement is the mean of a group of samples, from which we selected only those results in which CPU usage, as reported by the operating system, was at least 97% of the elapsed time; the resulting groups each contain at least five samples. We also exclude every result in which a protocol timer expired, since expiration of these timers causes retransmission after arbitrary intervals.

The largest standard deviation in any group was less than 2.5%; the maximum deviation from the mean was no more than 4.0%. This variability is of about the same magnitude as that reported for other computer performance tests, such as those reported in [Landherr86]. Note, also, that the measurements in this section have all been converted to appropriate performance rates.

5.1. Baseline Version Performance

By the *baseline version*, we mean the original Ada implementation as reported in [Biggar87], unmodified except for the inclusion of the single-node loopback paths. The results given in this and subsequent sections do not include any measurements for the ALS/N ADAVAX because this compiler (ALS/N ADAVAX Version 3.99) was unable to compile some of the packages in the baseline version. This problem has been reported to the ALS/N support organization, which is currently developing a solution; this has been identified as a problem with generics, whose fix, at the time of this writing, is scheduled for the December 1987 release.

Compiler	Simultaneous Connections		
	1	3	5
DEC Ada	5.8	5.4	—
VADS	4.2	4.1	4.2

Figure 9. Parallel Connection Rate (connections/sec)

Figure 9 shows the connect/disconnect rate for each of the compilers under test. Since the baseline Ada implementation represents each FSM as a separate task, we include a measurement of how the management of concurrent tasks affects the connection rate. In Figure 9, we show the results of this measurement. In this test, the driver opens a number of connections before closing any of them; during the connection establishment and disestablishment, therefore, a number of FSMs are operating simultaneously. Note that each connection in Figure 9 represents two FSM tasks.

The case of five simultaneous connections for the DEC compiler is excluded, because nearly all samples of this case involved protocol timeouts.

Compiler	Simultaneous Connections		
	1	3	5
DEC Ada	14.2	13.9	13.6
VADS	8.4	8.6	8.6

Figure 10. Effect of Parallelism on Data Throughput (Kilobytes/sec)

Compiler	Simultaneous Connections		
	1	3	5
DEC Ada	17.9	17.4	17.1
VADS	12.0	12.3	12.3

Figure 11. Effect of Parallelism on Message Throughput (messages/sec)

Figures 10 and 11 show the data throughput rate and message handling rate of the baseline implementation. Each figure was obtained by sending and receiving 150 messages, of the two sizes mentioned above, divided among differing numbers of simultaneous connections.

Our main use of the baseline performance measurements was to evaluate the effect of the variations described below. There are a few other points brought out by these measurements, however.

First, it is clear that the code generated by the DEC compiler outperformed that of the Verdix compiler. Part of this result is due to different compile-time options. In compiling under DEC Ada, we used an option that suppresses all of Ada's run-time checks. While VADS has a similar option, we found that using it caused the compiler to generate incorrect code; this problem has been reported to Verdix. Accordingly, the VADS code was measured with all run-time checks in place. For both compilers, we used as much optimization as the compilation systems provided. Our choice of options was governed by our desire to provide the fastest performance of which each compiler was capable.

To estimate the effect that the suppression of run-time checks has on performance, the reader may compare the results of the Dhrystone benchmarks, reported above. Also, Figure 12 shows a comparison of the performance of DEC Ada with run-time checks, DEC Ada without such checks, and VADS with the checks. The measurements shown in this figure are those of serial connection rate and of throughput of 150 messages over a single connection.

Compiler	Connection Rate (conn/sec)	Data Rate (Kbytes/sec)	Message Rate (msg/sec)
DEC (no checks)	5.8	14.2	17.9
DEC (checks)	5.2	11.6	15.6
VADS (checks)	4.2	8.4	12.0

Figure 12. Effect of Run-Time Checks

Another observation that may be made from the above data is that the use of simultaneous connections, i.e., simultaneous tasks, has little influence on performance. This observation must be used with care, however. For one thing, the above measurements do not include the overhead of creating and destroying tasks; our benchmark experience indicates that this time can be considerable.

Another piece of information not shown by these data is that the total number of tasks needed to be kept rather small. When the number of tasks was increased to, say, 50 or 100, as opposed to the numbers shown above, we experienced difficulties with the Ada run-time systems. These difficulties were experienced with both DEC Ada and VADS, and took the form of the test scenarios' failing to run to completion. In both run-time systems, even when run-time checks were enabled and when every task was given an exception handler, the errors took the form of failure to make progress; neither system raised an Ada tasking exception, or, indeed, any other exception.

5.2. Task Elimination

In this variation, as many tasks as possible were eliminated from the baseline implementation. Our goal was to eliminate as many rendezvous as possible from message processing, in order to determine the effect of rendezvous on throughput.

All TCP state machines and all IP logic were combined into a single task. This single task also included the ULP. Thus, nearly all of the code was combined into this single main task. Some other tasks were also eliminated. For example, the connection manager's synchronization task was eliminated, as was the IP identification task. Another IP task, which served to monitor incomplete message fragments for timeout action (i.e., essentially an IP garbage collector), was also eliminated. Overall, the number of rendezvous required to send and receive a single message over the IP loopback path was reduced from six to two.

The elimination of much of the tasking logic was facilitated by the modular design of both the protocol specifications and of this implementation of them. For example, all of the data required by a TCP state machine is specified as belonging to a single data structure called the state vector. It did not prove possible, however, to eliminate all tasks without changing the interface requirements between TCP/IP and its upper and lower layers. An important lesson we learned from this effort is that once a top-level design has expressed concurrency in terms of tasks, that decision becomes very hard to undo.

Compiler/	Simultaneous Connections		
variation	1	3	5
DEC/baseline	5.8	5.3	—
DEC/taskless	8.1	7.7	—
% change	+40 %	+45 %	—
VADS/baseline	4.2	4.1	4.2
VADS/taskless	5.4	5.0	5.3
% change	+29 %	+22 %	+26 %

Figure 13(a). Connection Rate, "Taskless" Variation (connections/sec)

Compiler/	Simultaneous Connections		
variation	1	3	5
DEC/baseline	14.2	13.9	13.6
DEC/taskless	18.5	18.8	18.6
% change	+30 %	+35 %	+37 %
VADS/baseline	8.4	8.6	8.6
VADS/taskless	10.0	10.0	10.0
% change	+19 %	+16 %	+16 %

Figure 13(b). Data Throughput, "Taskless" Variation (Kilobytes/sec)

Compiler/ variation	Simultaneous Connections 1	 3	 5
DEC/baseline	17.9	17.4	17.1
DEC/taskless	24.1	24.3	24.3
% change	+35 %	+40 %	+42 %
VADS/baseline	12.0	12.3	12.3
VADS/taskless	15.0	15.1	15.0
% change	+25 %	+23 %	+22 %

Figure 13(c). Message Throughput, "Taskless" Variation (messages/sec)

As can be seen from the measurements, the elimination of tasks has a profound effect on protocol performance. In fact, this variation had the largest effect of any of our experiments. The inference to be made from these measurements is that, at least for the compilation systems investigated here, tasks should be used sparingly when high performance is required.

Comparisons of the two compilers based on these data should be made with caution. Note that the fact that one compiler shows more improvement than the other does not indicate that it offers better performance in this area. It shows, instead, that the run-time system is less efficient in task management.

5.3. Record Representation

A significant aspect of protocol standards is the exact format of messages as they appear on external media, i.e. communication lines. Message formats must be very carefully specified to preserve the interoperability of different implementations of the protocol. In our baseline version of TCP/IP, arithmetic constructs are used to convert, back and forth, between the external media format of messages and the internal data structures used by the protocol code. These constructs have the advantage of being highly portable, but they have the disadvantage of generating relatively expensive code, compared to the bit-manipulation operators available on most computers.

This variation uses Ada record representation clauses to express the format of messages as they appear on external media. Such clauses are included in the Ada language to handle exactly the sort of problem posed by protocol interoperability, and so are well suited to this job.

Unfortunately, however, these constructs are not portable. The Ada standard permits Ada compilers considerable freedom in choosing the semantics of record representation (see [DoD83b, § 13.4]), with the result that record representation clauses do not have the same meaning from one compiler to another.

Since we had little doubt that record representation clauses would lead to a performance improvement, the purpose of this experiment was to quantify the improvement. Our goal was to understand the size of the tradeoff between improved performance and lower portability.

As might be expected, record representation had a negligible influence on connection speed, because these measurements do not include any IP formatting, which is the bulk of the formatting effort in the TCP/IP suite. The measurements for data handling are shown in Figure 14.

Compiler/ variation	Simultaneous Connections 1	 3	 5
DEC/baseline	14.2	13.9	13.6
DEC/rec. rep.	14.9	14.5	14.1
% change	+5 %	+4 %	+4 %
VADS/baseline	8.4	8.6	8.6
VADS/rec. rep.	8.9	9.0	9.0
% change	+6 %	+5 %	+5 %

Figure 14(a). Data Throughput, Record Representation Variation (Kilobytes/sec)

Compiler/ variation	Simultaneous Connections 1	 3	 5
DEC/baseline	17.9	17.4	17.1
DEC/rec. rep.	19.1	18.1	18.1
% change	+7 %	+4 %	+6 %
VADS/baseline	12.0	12.3	12.3
VADS/rec. rep.	12.7	13.1	13.1
% change	+6 %	+7 %	+7 %

Figure 14(b). Message Throughput, Record Representation Variation (messages/sec)

As can be seen from these data, the performance advantage gained by record representation clauses is only slight. It would therefore seem that they should be avoided, at least in code that is intended for use on heterogeneous systems. The remaining reason to use these clauses is that they offer greater conceptual clarity than arithmetic constructs. That is, the actual message layout can be seen as a static declaration, rather than inferred from dynamic code. This is partially offset by the fact that different compilers number bits differently.

5.4. Timer Tasks

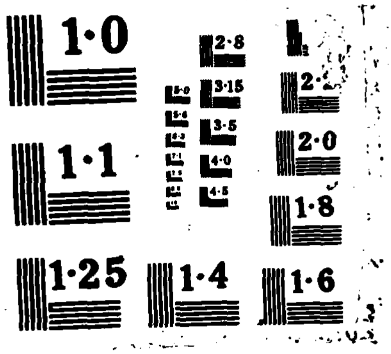
In most protocol specifications (e.g., [illegible] [DoD83a], [NBS81]), timers are represented as abstract objects with the following properties:

- They may be *set* to a certain time value.
- They *signal* when that time has expired [illegible] arbitrary value (i.e., a [illegible]
- They may be *canceled* or disabled [illegible]

Most protocol specifications [illegible] number of such objects. W[illegible] handful of timers [illegible] Bureau of Standards [illegible]

AD-A190 936   PROCEEDINGS OF THE NATIONAL CONFERENCE ON ADA (TRADE   3/7
MARK) TECHNOLOGY (6TH) HELD ON 14-18 MARCH 1988 IN
ARLINGTON VA(U) NORFOLK STATE UNIV VA MAR 88

UNCLASSIFIED   F/G 12/5   NL


1·0
2·8
3·15
3·5
4·0
4·5
2·2
2·0
1·1
1·8
1·25
1·4
1·6

hundreds. It has been pointed out [Castanet86] that Ada tasks can model such objects very well. This is a relatively expensive way of managing time in Ada, however. A more natural way is to use timed entry calls [DoD83b, § 9.7.3] or selective waits [DoD83b, § 9.7.1]; our baseline implementation uses the latter. This variation measures the tradeoff between this method and the more flexible method provided by timer tasks.

Compiler/	Simultaneous Connections		
variation	1	3	5
DEC/baseline	5.8	5.4	—
DEC/timers	5.1	4.3	—
% change	−12 %	−20 %	—
VADS/baseline	4.2	4.1	4.2
VADS/timers	4.1	3.8	3.9
% change	−2 %	−7 %	−7 %

Figure 15(a). Connection Rate, Timing Tasks Variation (connections/sec)

Compiler/	Simultaneous Connections		
variation	1	3	5
DEC/baseline	14.2	13.9	13.6
DEC/timers	12.7	12.2	12.1
% change	−11 %	−12 %	−11 %
VADS/baseline	8.4	8.6	8.6
VADS/timers	8.1	8.3	8.3
% change	−4 %	−3 %	−3 %

Figure 15(b). Data Throughput, Timing Tasks Variation (Kilobytes/sec)

Compiler/	Simultaneous Connections		
variation	1	3	5
DEC/baseline	17.9	17.4	17.1
DEC/timers	15.6	15.1	15.0
% change	−13 %	−13 %	−12 %
VADS/baseline	12.0	12.3	12.3
VADS/timers	11.4	11.7	11.7
% change	−5 %	−5 %	−5 %

Figure 15(c). Message Throughput, Timing Tasks Variation (messages/sec)

Figure 15 shows the results of this experiment. It can be seen that the overhead for timer tasks was relatively high, at least for one of the compilers, leading to the tentative conclusion that this method of time management should be avoided.

It should be kept in mind, however, that Ada selective waits implement a slightly different timer model than the one described above. Although an Ada selective wait can have more than one **delay** branch, the expiration of any one of these branches effectively cancels the waits on all the others. This is suitable for protocol specifications in which all pending timers are cancelled when any one of them expires, but it may not be well suited to protocols in which this is not done.

Once again, it can be seen that one compiler is more profoundly affected by the number of running tasks than the other. This corresponds to the results noted for the task elimination variation, described above.

6. Comparison of Ada and C Implementations

Part of our overall evaluation of the suitability of Ada as a protocol implementation language was to compare a protocol implementation written in Ada to the same protocol implemented in C. The C implementation we chose to measure was written by Michael Wingfield and Tom Blumer of Bolt, Beranek, and Newman (BBN) in 1980, under contract to the DCA [Wingfield80]. This implementation is currently used by several ARPANET hosts.

The reader should be aware, however, that the BBN implementation does not conform to the full military standard TCP and IP. The BBN implementation is written to conform only to preliminary standards for these protocols [Postel80b, Postel80a], which contain fewer state transitions than in the current standards. The Ada implementation, on the other hand, conforms completely to the current standards. This has been confirmed by subjecting this implementation to the DCA's protocol validation test suite [Mankin87, Griffin86].

6.1. *Structure of C Implementation*

Our comparative measurements were made with the BBN implementation relatively intact, that is, with only the changes necessary to introduce the single-node loopback capability. Since the overall structure of the BBN implementation differs markedly from that of our Ada implementation, we shall outline the BBN approach below.

The BBN implementation is designed to be a stand-alone, continuously running protocol *server* which accepts commands from other programs under the control of the UNIX operating system. This program makes use of an interprocess communication (IPC) mechanism known as "Rand" ports [Sunshine77, Zucher77], an extension to UNIX developed by the Rand Corporation and BBN. This IPC facility is not available at most UNIX sites. Berkeley UNIX, however, now offers an IPC facility with the same functionality as the Rand ports. Accordingly, we augmented the BBN implementation with a set of library routines that convert the Rand port calls to the appropriate Berkeley UNIX calls.

To drive the BBN implementation, we used the same Ada *driver* package that we used in our other tests. This package was modified to be a separate program, by replacing the connection manager package with a set of calls on an interface library to send and receive IPC messages to and from the server program. The interface library is supplied as part of the BBN implementation of TCP/IP. Since we ran the BBN implementation in a UNIX environment, the driver program was compiled with the VADS compiler.

8.2. Comparison Results

The most difficult part of comparing the BBN implementation to the Ada implementation is to avoid including any differences that are due only to the radical difference in their overall architecture. In order to take this difference into account, we present two measurements for each test of the BBN implementation.

First, we present performance rates based on the mean value of total elapsed time. The reliability of these measurements is about the same as those in the previous section, with all standard deviations less than 5%. Second, we present performance rates with IPC time excluded. These rates are based on CPU usage, excluding any time spent in the IPC library routines. This measurement has a greater statistical fluctuation, with some standard deviations as high as 18%.

Each measurement presented in this section was obtained on systems that were free of other users, and is based on at least five samples.

Compiler/	Simultaneous Connections		
variation	1	3	5
DEC Ada	5.8	5.4	—
VADS	4.2	4.1	4.2
DEC/taskless	8.1	7.7	—
VADS/taskless	5.4	5.0	5.3
BBN (elapsed)	2.3	2.7	2.5
BBN (CPU)	7.6	7.5	7.5

Figure 16. Connection Rate, BBN C Implementation (connections/sec)

Figure 16 shows the connection establishment and disestablishment time for the BBN implementation. For ease of comparison, the figure also shows the performance of the Ada baseline versions and the Ada reduced-task variations. It can be seen that performance of the C implementation is in the same neighborhood as that of the Ada implementations.

Compiler/	Simultaneous Connections		
variation	1	3	5
DEC/baseline	14.2	13.9	13.6
VADS/baseline	8.4	8.6	8.6
DEC/taskless	18.5	18.8	18.6
VADS/taskless	10.0	10.0	10.0
BBN (elapsed)	7.2	7.1	—
BBN (CPU)	10.3	10.2	—

Figure 17. Data Throughput, BBN C Implementation (Kilobytes/sec)

In this test, the performance of the BBN implementation is also in the neighborhood of most of the Ada implementations. It is worth noting, however, that the best Ada implementations clearly outperform the BBN implementation, *even when IPC time is excluded.* (The measurement for the BBN implementation with five simultaneous connections is missing from this table because the limitations of the 4.3 BSD IPC system did not permit this test to be run.)

Compiler/	Simultaneous Connections		
variation	1	3	5
DEC/baseline	17.9	17.4	17.1
VADS/baseline	12.0	12.3	12.3
DEC/taskless	24.1	24.3	24.3
VADS/taskless	15.0	15.1	15.0
BBN (elapsed)	15.1	14.5	14.3
BBN (CPU)	31.1	32.6	30.1

Figure 18. Message Throughput, BBN C Implementation (messages/sec)

Finally, in our test of message handling rate, in which the messages contain only one character of user data, the performance of the BBN implementation is less than that of the best Ada implementations. When IPC time is excluded, the BBN performance is greater than that of any Ada implementation. Note that almost all of the data exchanged in this test consists of message headers.

Overall, the measurements shown above show that an interactive user of these protocols would find the Ada performance at least as good as that of the C implementation. For applications where the message size is large, such as file transfer or electronic mail, the Ada performance, under some Ada compilers, would be far better.

Our experience with porting the BBN implementation to the VAX from its original host machine, a DEC PDP-11, illustrates an important aspect of this implementation other than simple execution speed. The BBN implementation proved to be extremely machine-dependent. Its principal machine dependency was the assumption of a 16-bit integer size. It had many other machine dependencies, however, including many places in which the size of data structures in bytes was included explicitly, even where the C **sizeof** operator could have been used. It also included assumptions about field alignment within records that, we discovered, are not common to all C compilers.

This contrasts markedly with the high portability of the Ada implementation, which was ported without changes to two different hosts (VAX and Sun) and two different operating systems (Berkeley UNIX, and VMS). Much of this portability is due to good Ada design, and, in particular, to the avoidance of system-defined types, such as INTEGER. Note, however, that the Ada language offers the implementor much more opportunity to achieve portability than does C.

7. Conclusions

Our most important conclusion is that protocol implementations in Ada can offer performance that is comparable to implementations written in C. We therefore conclude that Ada has demonstrated its capability to handle real-time applications.

Our experience does show that there is wide variation in the performance offered by different Ada compilers. We notice, however, that the difference in protocol performance is not necessarily reflected by our benchmarks of individual features. This indicates that all language constructs do not contribute equally to overall program performance. We would therefore suggest that the vendors of Ada compilers concentrate on applications-oriented benchmarks when attempting to improve performance, rather than on single-feature benchmarks.

Our Ada protocol tests show that the run-time systems offered by current Ada compilers can handle only limited concurrency. To meet the needs of the communication protocol domain, the robustness of these systems will need improvement.

The Department of Defense is in the process of requiring the use of Ada in all future software. Some have held, however, that Ada's performance was unsuited to real-time applications, such as communications protocols. We have shown that this is no longer the case. Existing, commercial Ada compilers can offer performance which competes with that of C in implementing DoD protocols. We therefore believe that the use of Ada in future protocol implementations should be encouraged by the Department of Defense.

8. References

[ALSN87] *Ada Language System/Navy Reference Handbook*, Raytheon Service Company, Arlington, Virginia, 30 June 1987. (Document No. ALSN-HBK-PSE-REFHB).

[Barnes84] J. G. P. Barnes, *Programming in Ada*, Addison-Wesley, Reading, Massachusetts, 1984.

[Biggar87] J. Biggar, "The DoD TCP/IP Protocol Suite in Ada," *Proc. 1987 Unisys Software Engineering Symp.*, McLean, Virginia, September 1987.

[Castanet86] R. Castanet, A. Dupeux, and P. Guitton, "Ada a Well Suited Language for Specification and Implementation of Protocols," in *Protocol Specification, Testing, and Verification, V*, M. Diaz (editor), North-Holland, New York, NY, 1986.

[DCA82] *DoD Protocol Reference Model*, System Development Corporation, Santa Monica, California, 30 September 1982. (DCA Contract No. DCA100-82-C-0036).

[DEC85] *VAX Ada Language Reference Manual*, Digital Equipment Corporation, Maynard, Massachusetts, 1985. (VAX/VMS Version 1.0).

[DoD83a] *Military Standard Transmission Control Protocol*, United States Department of Defense, 12 August 1983. (MIL-STD-1778).

[DoD83b] *Reference Manual for the Ada Programming Language*, United States Department of Defense, 17 February 1983. (ANSI/MIL-STD-1815A).

[DoD83c] *Military Standard Internet Protocol*, United States Department of Defense, 12 August 1983. (MIL-STD-1777).

[Griffin86] B. Griffin, "DCEC Protocol Laboratory Internet Protocol Certification Test Index," Unisys Report No. TM-8801/101/00, 6 November 1986.

[Hook85] A. A. Hook, G. A. Riccardi, M. Vilot, and S. Welke, *Ada Compiler Evaluation Criteria*, Institute of Defense Analyses, October 1985.

[Hughes86] *Ada Benchmark Suite, Tasking Section*, Hughes Aircraft Company, Ground Systems Group, Software Engineering Division, San Diego Software Engineering Laboratory, Command and Control Software Department, 29 August 1986.

[ISO83] *Information Processing Systems – Open Systems Interconnection – Basic Reference Model*, International Standards Organization, May 1983. (Draft International Standard ISO/DIS 7498).

[Landherr86] S. Landherr, J. Kochmar, and A. Sun, *Evaluation of Ada Environments: Chapter 8 – ACEC Tests*, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1986. (SEI-86-MR-10).

[Mankin87] A. Mankin, "TCP Traceability Matrix," Unisys Report No. TM-8801/206/00, 6 February 1987.

[NBS81] *Specification of the Transport Protocol, Volume 3: Extended Class Protocol*, United States National Bureau of Standards, September 1981. (NBS Report No. ICST/HLNP-81).

[Postel80a] J. B. Postel, "DoD Standard Internet Protocol," IEN 128, Information Sciences Institute, January 1980.

[Postel80b] J. B. Postel, "DoD Standard Transmission Control Protocol," IEN 129, Information Sciences Institute, January 1980.

[Squire85] J. Squire, "Performance Issues Workshop," *ACM SIGAda Users Committee Performance Issues Working Group*, 15-16 July 1985.

[Sunshine77] C. Sunshine, "Interprocess Communication Extensions for the UNIX Operating System: I. Design Considerations," R-2064/1-AF, Rand Corporation, June 1977.

[Tanenbaum81] A. S. Tanenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Verdix86] *Verdix Ada Development System*, Verdix Corporation, Chantilly, Virginia, 1986. (Sun3/UNIX Version 5.4).

[Weicker84] R. P. Weicker, "Dhrystone : A Synthetic Systems Programming Benchmark," *Comm. ACM*, **27**(10) (October 1984), pp. 1013-1030.

[Wingfield80] M. A. Wingfield, "TCP Software Documentation," BBN Report No. 4295, Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts, February 1980. (prepared for the Defense Communications Engineering Center, contract no. DCA100-78-C-0011).

[Zucher77] S. Zucher, "Interprocess Communication Extensions for the UNIX Operating System: II. Implementation," R-2064/2-AF, Rand Corporation, June 1977.

Robert H. Pollack
P.O. Box 517
Paoli, PA 19301-0517

Mr. Pollack has over nineteen years experience in programming and software support, on a variety of hardware and operating systems. He has been responsible for several protocol implementations, including IP and the NBS transport protocol. He is currently principal investigator of a research project in the automatic creation of programs from specification languages, using generalized compiler-generation technology implemented in Ada. He has applied this technology to create several significant Ada systems. Mr. Pollack holds an M.S.E in computer science from the University of Pennsylvania.

David J. Campbell
P.O. Box 517
Paoli, PA 19301-0517

Mr. Campbell has over twelve years experience in compiler, operating system, and support tools development. He is part of the Unisys Ada compiler development team, and also has done extensive work on the automatic generation of Ada code for application-specific domains. He has implemented several different variations of a reference protocol suite written in Ada. Mr. Campbell holds a B.S in computer science from Wichita State University.

ADA TOOLS FOR THE DESCRIPTION AND SIMULATION OF DIGITAL SIGNAL PROCESSING SYSTEMS

Mark D. Happel

Northern Telecom Inc.
Research Triangle Park
North Carolina

Brian E. Petrasko

EECS Department
University of Central Florida
Orlando, Florida

ABSTRACT

While specialized hardware description languages allow for maximum capability and efficiency in a design automation system, the use of a general purpose language in the same role can make the system more available or more practical for a larger set of users. This project demonstrates the use of ADA* for the description and simulation of small digital signal processing systems. Building on conventions and primitives proposed by Denyer and Renshaw, a simple subsystem was described in ADA and then tested with a small simulator also written in ADA.

INTRODUCTION

In order to deal with the complexity problem facing integrated circuit and system designers, structured design methodology and practices are now being applied to electronic design. Concepts of modularity, block-structuring, system hierarchies and structured descriptive techniques that have been utilized in large programming projects have been successfully applied to VLSI design (Lea 1986).

The manner in which digital logic circuits are represented has been strongly influenced by these structured design tools. In the 1950s, I.S. Reed developed a descriptive notation for bit transfers in sequential digital systems that became known as a register transfer language (Dietmeyer and Duley 1975) and has since been expanded to encompass a number of so-called hardware description languages. The hardware description languages now available differ widely in syntax, features and capabilities (Aylor, Waxman and Scarratt 1986) and are used primarily to provide structured descriptions of hardware and to allow for the computer simulation of the description in order to confirm the intended behavior of the circuit at various levels (Dietmeyer and Duley 1975).

Toward a Common Language

Unfortunately, utilization of these descrip*tive languages has* been hampered by the lack of a clear standard. Many of the languages have remained in purely academic applications, while the remainder tend to be the property of a particular manufacturer and not available to companies without the resources to develop their own software tools. Worse, the lack of a standard places a severe limitation of the interoperability of various computer aided engineering tools and environments (Dewey and Gadient 1986).

In response to the lack of standards, the Department of Defense (DOD) has undertaken a project to develop a standardized hardware description language as a part of its very high speed integrated circuit (VHSIC) program. This language has been named VHDL, for VHSIC hardware description language (Waxman 1986). The VHDL language is both an extension and a subset of the general purpose programming language ADA* (Wallace 1986), a language developed during the 1970s and early 1980s by the Department of Defense as a standard language for the programmine of embedded computer systems (Chohn 1986).

One of the most controversial aspects of VHDL is the extent of the similarity between VHDL and ADA. During reviews of the VHDL standard, some have argued for the full inclusion of ADA in VHDL, while an equal number have fought against too much similarity. Those arguing for the inclusion of ADA maintained that "since VHDL will be implemented in ADA and the behavioral component will describe behavior, ADA seems the simplest and most robust means to provide structured hardware design". Those arguing against the inclusion of *ADA* felt that since VHDL is a hardware description language and not a general programming language, the inclusion of ADA would over-complicate the VHDL language, making learning the language overly difficult (Nash and Saunders 1986).

Despite the potential standardization offered by VHDL and similar efforts, it is quite likely that the implementation of a single set of standard tools capable of the engineering of chips for all application areas should not be expected anytime in the near future. Efforts at the present time appear to be concentrating on systems that deal with only one application area in particular (DeMan et al. 1986).

* ADA is a trademark of the United States Department of Defense - ADA Joint Program Office (AJPO).

Digital signal processing has been a popular applications area among design automation developers. Due to the highly specialized, complex algorithms and high data throughput rates, custom ICs have become a necessity in signal processing applications (DeMan 1986). A notable example of work in this area is the system developed by Denyer and Renshaw (1985) of the University of Edinburgh. This system, a so-called silicon compiler, takes a description of the desired behavior of the circuit and produces as output the various masks required for chip fabrication. The system also provides for verification of the design via simulation of the behavioral description.

The Denyer and Renshaw project utilizes a special purpose language specifically developed for that system. This language, "FIRST," (Denyer and Renshaw 1985) has been developed from the ground up with hardware description in mind. Some designers of hardware description languages feel that the similarity often observed between programming languages and description languages is unfortunate and the result of the lesser evolution of description languages compared with that of programming languages (Dietmeyer and Duley 1975).

An Alternate Approach

While special purpose languages can be optimized for an individual task or set of tasks, they also present inherent disadvantages. In order to use a new description language, one must obviously become familiar with it. Perhaps not as obvious is the need for proper support for this language -- compilers, debuggers, development tools, etc. This additional required support adds to the total software overhead of any project, and may prevent the use of other tools or software support not written for that special purpose language.

There has been interest in and study of the use of general purpose programming languages to describe and simulate hardware. It has been suggested that strong typing, automatic memory management and polymorphic operators be features of languages to be used in this role (Ayres 1979). Certainly, the power and versatility of recent programming languages could prove useful for hardware description.

Purpose of Work

The purpose of this work was to examine the potential of the ADA programming language for the description and subsequent simulation of bit-serial digital signal processing architectures. ADA was chosen for examination due to its versatility, standardization and DOD sponsorship.

This work was carried out by encoding in ADA several of the fundamental architectural blocks, or "primitives," selected by Denyer and Renshaw. Once encoded, the primitives were collected, along with important simulation routines, into an ADA package. A simple simulator was written to verify a description of a DSP algorithm in terms of DSP primitives and other hierarchical blocks.

The final project step was to create the description of a modest digital signal processing subsystem to exercise the primitive and simulation routines. As an initial example, a complex number to magnitude conversion unit was chosen. This particular example has been demonstrated by Denyer and Renshaw using their specialized hardware description language, and therefore, made an interesting case study for the use of ADA (Denyer and Renshaw 1985).

ALGORITHMIC VLSI DESIGN

A Building Block Approach

In 1980, a text was published (Mead and Conway 1980) that has had a significant impact on both the present and future of VLSI design. In this text, Mead and Conway advocate a VLSI design *technique of functional blocks interconnected to form larger systems* (Lea 1986). Each of these functional blocks can be individually designed as a separate circuit and eventually reduced into *integration masks*. Once the circuit, or "leaf-cell," corresponding to a functional block has been designed, it can be used as a building block, with more complex integrated circuits being formed by the *interconnection of the leaf-cells*. Different VLSI chips can be developed by interconnecting various standard leaf-cells, without requiring the redesign of the leaf-cells themselves (Mead and Conway 1980).

This design style for VLSI chips has been labelled the "simplified full-custom leaf-cell design style" (Lea 1986) and has led to the development of standard *cell blocks. The approach* sacrifices the efficiency (in terms of silicon area and performance) of a fully optimized design in favor of a more structured hierarchical approach. This structured approach, similar to the top-down design methodology popular in large software projects (Fairley 1985) reduces the design complexity of large systems to a manageable level by allowing the designer to concentrate his efforts on one set of problems at a time, omitting complicating details of lower hierarchical levels. The similarity to software engineering methodology has made this approach popular among developers of silicon compilers (Lea 1986).

The Denyer and Renshaw Project

It is this standard cell approach that has been adopted by Denyer and Renshaw for their silicon compiler research. In this case, the standard cell is a generic bit-serial processing element. The use of bit-serial communications between the cells permits simplification of the networks connecting the cells, a potential problem in the chip layout (Denyer and Renshaw 1985).

The hierarchy used in this project is shown in Figure 1. All design levels below the leaf-cell level have been automated into a silicon compiler and are, therefore, not of concern to a designer specifying a digital signal processing system. The DSP designer can view the leaf-cells as functional "black boxes."

A base set of leaf-cells for DSP, called primitives, was developed by Denyer and Renshaw. A description of a DSP system written in terms of these primitives is input to a silicon compiler which would produce as output the masks for making integrated circuit chips to perform the DSP function. When processing the description, the silicon compiler calls routines corresponding to the individual primitives involved and mask sections for the leaf-cells in appropriate areas of the chip are produced.

In order to check the description before the time and expense of silicon compilation, the description should first be processed by a behavioral simulator. As with the silicon compiler, the simulator calls routines corresponding to the description primitives. However, in this case, the routines simulate the behavior of the primitives by processing sample test data and simulating propagation delays through the system. By comparing the simulation data with the desired output, design flaws can be detected prior to silicon implementation.

This paper concerns the simulation of the DSP system descriptions. The DSP descriptions have been developed to be consistent with Denyer and Renshaw's format and timing conventions (Denyer and Renshaw 1985). Specifically:

1. Data is fixed point, bit-serial, LSB first.
2. The primitives are operated synchronously from a master two-phase clock.
3. Control signals are delayed by an integer number of clock cycles via an associated control network to provide primitive synchronization.
4. Each operator possesses a fixed latency (propagation delay from input to output) which is also an integer number of clock cycles. (Since operation is bit-serial, with one bit of data entering or leaving a primitive during each clock cycle, the terms "bit" and "clock cycle" will be used interchangeably when referring to timing considerations.)
5. Each primitive operator has an optional, one bit input pre-delay in order to reduce the number of single bit delay leaf-cells that might otherwise be required to ensure that all necessary input data arrives simultaneously).

System

Chip

Operator

Primitive
(Leaf-cell)

Specified by DSP Designer

Automated-Implemented by Silicon Compiler

Register-Transfer Circuits

Gate-Level Circuits

Semiconductor Devices

Integration Masks

Figure 1. Digital Signal Processing Hierarchy.

A SAMPLE ALGORITHM

The algorithm chosen to exercise the ADA behavioral description capabilities, as well as the operation of the accompanying simulator, is a relationship that approximates the magnitude of a complex number. This function, called the "four region approximation," calculates the magnitude, M, of a complex number, I + jQ, by (Filip 1976):

$$M = \mathrm{MAX}\begin{pmatrix} (I) \\ ((7/8)\,I + (1/2)\,Q) \\ ((1/2)\,I + (7/8)\,Q) \\ (Q) \end{pmatrix}$$

Denyer and Renshaw proposed the following restatement of the four region approximation to improve its structure for implementation as:

$$M = \mathrm{MAX}\begin{pmatrix} G \\ ((7/8)G + (1/2)L) \end{pmatrix}$$

where G = max (I, Q) and L = min (I, Q) (Denyer and Renshaw).

A flow graph based on the above algorithm is shown in Figure 2. Note that the numbers within circles represent delay times for synchronization of data and control signals. The control signal C1 arrives simultaneously with the LSBs of the real and imaginary input data words. The notation C1-X refers to the C1 control signal delayed by X bits.

The arcs interconnecting the primitives are designated by an ID number in parentheses. Labels are also used where appropriate. The latency associated with each primitive is shown in brackets below the primitive name.

Note that "seven-eighths" is not a primitive as such, but is actually an operator made up of primitives in accordance with the hierarchy shown in Figure 1.

BEHAVIORAL DESCRIPTION IN ADA

Descriptive Levels and Abstraction

The intent behind digital hardware description in a high level language is to permit the precise specification of the intended structure, capabilities, and/or functions of the system being described. This description could be an abstract algorithmic specification with no implication of actual hardware structure, a detailed gate-level description that obscures the system behavior, or something in between these two extremes. In addition, different hierarchies are possible, with higher levels using more powerful building blocks like adders and multipliers instead of the lower levels' combinatorial logic gates.

For the purpose of this project, it was decided to limit the description to levels at or above the primitive level of the hierarchy shown in Figure 1, with a fair degree of abstraction. The ultimate intent of a system such as this is to allow a system designer to specify an algorithmic description of the system in terms of primitives like multiply and absolute value, leaving the development of the details of lower levels to the automated silicon compilation process and thus freeing the DSP designer from the unnecessary details of low level digital synthesis. Thus, the designer can concentrate on the intended behavior and need not be a skilled digital designer or MOS circuit designer.

It is also true that the strong data-typing and verbosity that makes ADA and similar structured languages more readable and easily documented than unstructured languages also tends to clarify behavioral descriptions while lengthening and perhaps overly complicating structural descriptions. It has been noted that while the ADA-derived VHDL has been designed for use at various levels of behavioral abstraction, it has proven much more successful at behavioral description than at structural description (Nash and Saunders 1986).

Primitive or Simulation Package Level

The abstraction of the description is strongest within the primitive themselves. For example, consider the statement:

```
ARC (OUTPUT).DATA := ABS(ARC(INPUT).DATA)
```

which takes the absolute value of the input signal arc and assigns it to the output signal arc. While this could be accomplished in hardware with shift registers and complementing logic (Denyer and Renshaw 1985), the description here implies virtually nothing about the actual structure. The only clue to the nature of the hardware lies in the calculation of latency as the sum of a constant plus the system word length, implying internal temporary storage of the incoming serial bits, as well as an additional processing delay. If it were not for the need to include numerous simulation details within the primitive, the function of the unit would be quite obvious.

Operator Description Level

The level of abstraction is reduced somewhat in the next hierarchical level which involves the description of the complex-to-magnitude operator. This is the lowest level which the designer need deal with, as the primitives have been provided for him as part of the package "SIMPKG." The level of involvement with the simulator has been minimized to just a requirement to place the actual description within a case statement, allowing conditional execution of primitive routines.

The description at this level consists primarily of procedure calls to various primitive subroutines. This is similar to the syntax of several descriptive languages such as Denyer and Renshaw's FIRST, where the routines are behavioral primitives (Denyer and Renshaw 1985) and Hill and Peterson's AHPL, where the routines are general purpose MSI integrated circuits (Hill and Peterson 1981). In AHPL, the use of hardware primitives keeps the level of abstraction in a system description low, while the behavioral primitives of this paper maintain a fairly abstract operator description.

Nevertheless, there is more structure implied at the operator level than at the primitive level. While the description can be regarded as a structured verbal specification of the complex to magnitude algorithm flow graph, it also tends to imply a physical communication network between the primitive "black boxes."

It should be noted that the verbosity of ADA so useful for software readability can be used to improve hardware description. In this case, a feature known as named association, which binds subroutine formal parameters with the calling routine's actual parameters via the syntax (Department of Defense 1983):

```
Formal Parameter   =>   Actual Parameter
```

allows the description:

```
Subtract (minuend => 9, subtrahend => 8,
          difference =>12, Latency => 1,
          Ctrl => C1(12));
```

instead of the shorter but less revealing description:

```
Subtract (9, 8, 12, 1, C1(12));
```

which requires examination of the primitive description or documentation to reveal the actual interconnections. The first three parameters (9, 8 and 12) are the numerical designations of the input and output arcs (type ARC_NO) satisfying the formal procedure specification:

```
Procedure Subtract (MINUEND, SUBTRAHEND,
                    DIFFERENCE: ARC_NO;
                    CTRL: NATURAL;
                    LATENCY: POSITIVE: = 1;
                    DEL1, DEL2: BIT: = 0);
```

Significantly, the use of named association is also an optional (but recommended) feature of VHDL (Lipsett, Harschner and Shahdad 1986).

Also, in keeping with another VHDL example, defaults are provided for some of the parameters in the primitive routines. This frees the designer from specifying connections to seldom used inputs or features, like the optional input predelay. While this is provided to hide unnecessary detail and streamline the description, it is recommended that it be used with caution as unintended omission of parameter specifications may result in operation other than that intended, with the root cause difficult to detect.

A useful feature of the description method developed here is that operators can be used as building blocks for more complex operators. Thus, a frequently occurring or useful operator, like complex-to-magnitude, may be used as a building block of an operator at the next higher hierarchical level, once again managing complexity by abstraction -- in this case, the inner workings of the complex-to-magnitude operator. Although it complicates the simulation somewhat, the added power of description was deemed well worth the additional trouble.

It should be noted that the control portion of the operator is separated from the data processing portion in keeping with the popular register transfer sequential machine model (Hill and Peterson 1981). This separation is emphasized by use of a block declaration for the control and a separate block declaration for the data unit. These block structures are provided solely to clarify the description and serve no useful purpose as far as simulation is concerned, though they may prove useful to leaf-cell or floorplan compilers operating on the same description.

The Chip Level

The highest level of description in the present project is the chip level, representing a single VLSI chip. This chip consists of the associated operators, as well as a master control generator for overall signal flow coordination and the input and output pads and buffers necessary to route data to and from the chip. While the Denver and Renshaw system has a higher, multi-chip system hierarchy (Denver and Renshaw 1985). As shown in Figure 1, limiting this project's description to single chips only was done to limit the scope of this work and is not perceived to be a limitation of ADA descriptions. It is felt that only minor changes would be necessary to the present simulator and primitives to allow for multiple chip systems.

Simulator Structure

Compared to the effort required to establish conventions for hardware description in ADA, the construction of a simulator for the ADA description proved a much more formidable and time-consuming task. The problem was to develop a simulator, part of which would be a descriptive routine (the "chip" description) that ideally would contain little or no direct reference to simulation variables or requirements in order not to burden the designer with simulation details, and yet still be efficient enough to perform its functions in a reasonable time with reasonable resources. Although the simulator finally developed is rather limited and perhaps somewhat simplistic, the limitations were a result of the scope of the project and, it is believed, could be expanded to a more sophisticated system without changing its basic structure or that of the hardware description discussed previously.

The simulator consists of a main routine "SIMULATE", which calls the description "SYSTEM" as a subroutine. The "SIMULATE" routine is actually little more than a user I/O interface, an initialization section and a loop containing the system description and a simulation time advancement routine. The "SYSTEM" routine is compiled separately and linked at run time, permitting the compiling and linking of different "systems" at run time without requiring recompilation of the main simulator routine. This maintains the general nature of the simulator.

Scope of Simulation Variables

As mentioned previously, it was considered desirable that as many simulation tasks to be performed by the system as possible. This led to many details being buried within the primitive routines that are called by the description routine. As these primitives are provided to the user via the package "SIMPKG," the user is spared from simulation technicalities. Unfortunately, the placement of the description routine on a software level above the primitive routine level but below the simulator level made it impractical to pass simulation variables, such as simulation time and data structures, via specified formal parameters since the simulation variables would have to be explicitly named in the parameter lists of the "system" call in the "simulate" main routine and the primitive calls in the system description. This limitation could only be overcome by extensive use of global parameters, a practice discouraged by software engineers due to susceptibility to accidental modification of variables or unforeseen side effects (Fairley 1985). ADA practitioners warn of obscured information passing and software maintenance difficulties (Cohen 1986). Despite their warnings, the use of global variables in this case appears warranted by the benefit of allowing the system designer to ignore the communication of simulation information between the simulator and the primitives. It is interesting to note that VHDL has been criticized for forcing the current simulator time to be passed via port (parametric) interconnections, hampering high level behavioral models that would profit from internal access to a global simulation time (Nash and Saunders 1986).

Data Structures

The fundamental data type in this project is a type defined in "SIMPKG" as type "signal." This is a record consisting of a floating-point data variable and an integer time variable. Thus, a signal becomes a two-dimensional vector tying sample data and time of occurrence. This is similar to the basic construction of another digital signal processing descriptive language, utilized in the Cathedral-II Silicon Compiler Project (DeMan et al. 1986).

The type signal is used to define a global array of signal-type variables that is called the arc array. This is a listing of signals passing between primitives (nodes of a signal flow graph). By passing the index numbers of different elements of the arc array to the primitives as parameters of the primitive calls (in the hardware description), the arc array can be used as a source of input data for primitives as well as storage for output data. By specifying the same index to one primitive as output and another primitive as input, the two primitives are effectively joined together just as the nodes of a signal flow graph are joined by arcs.

Simulation Abstractions

It must be noted that some abstractions have been introduced into the simulation to improve throughput. Most significantly, the simulation primitives operate on parallel words (not bit-serial as in the actual hardware) that occur at the same time as the earliest (least significant) bit of the serial data word. By using parallel data in this fashion instead of one serial bit at each clock time, the primitive can process the entire word during the same clock (simulation) time and eliminate the need to call the primitive routine at every click of the simulation clock. By adding adjustments to the primitive's internal algorithmic description, the same bit-fidelity as the actual bit-serial approach can be maintained while providing a significant increase in simulator throughput (Denyer and Renshaw 1985).

Another deviation from the actual hardware convention was the use of floating-point variables for the system data instead of the fixed point data in the actual hardware. Here, floating-point variables were used due to the ease with which ADA operates on floating-point variables compared to fixed-point quantities. Further work along the lines of this project should use fixed-point data to more accurately model the limitations of the hardware, though for this project it is not believed that the use of floating-point data significantly contaminated or invalidated the results.

The Event Queue

Due to the pipelined architectures of the signal processing systems under study, where new input data enters the system before the previous data has been completely processed in order to improve throughput, the arc array proved unsatisfactory as the sole means of storing the data on the arcs. Evaluation of timing diagrams for the system revealed that output data could be overwritten by later outputs before being input to the following primitive if the preceding primitive had a latency greater than one word length. The use of multiple arc arrays and status bits was discarded as too cumbersome.

In order to solve the data overwriting problem, it was decided to load the output data, the output time (equal to input time + pre-delay + latency) and the arc index (identifying the arc) onto a queue whenever a primitive completes its operations on a signal. The arc index-output data-output time vector can now be referred to as an event. Events are stored in the queue in order of increasing event time with the earliest occurring event at the head of the queue. When it is time for a primitive to process a data sample, it locates the data in the queue by locating the desired input arc index and event time and then loading the corresponding data into the proper location in the arc array. Thus, each primitive is assured of valid input data and the overwriting of data of the arc array is immaterial. Indeed, the arc array now becomes more of a communications path and less of a storage facility, with the *latter role now being performed* by the queue. Overwriting in the queue is avoided by defining a new record for every new event, discarding it only after loading it back into the arc array on the request of a primitive routine.

The queue is implemented as a global recursive-style linked list. Event records are dynamically allocated during run-time and linked via access types (similar to pointers in C or PASCAL) to other event records already in existence. Events are loaded onto the queue by the SIMPKG procedure "ENQUEUE," returned to the arc array by the procedure "RETRIEVE," and deleted from the queue by the procedure "DELETE."

Advancing Simulation Time

The storage of data on the queue serves a much more useful purpose than just ensuring the validity of primitive input data. Since the events in the queue are ordered by increasing time, the event at the head of the queue will be the next event to occur in the simulation. If the time of occurrence of the next event is greater than the current simulation time (i.e., the next event occurs at some time in the future), it would be pointless to invoke any new primitives, as no new data would be available for processing. The same would hold true for all simulation times earlier than the next event time. Obviously, the simulation clock should skip ahead to the time of the next event, ignoring intermediate clock cycles.

This results in an event-driven simulation and a significant improvement in simulator throughput. It is similar, in fact, to Denyer and Renshaw's event-driven behavioral simulator (Denyer and Renshaw (1985).

Control Considerations

Control signals are generated in a chip description by delaying the outputs of the chip's "control generator" primitive. The delays are accomplished by the "CBITDELAY" primitive, which delays the original control signal by a user-specified latency. There are three possible control signals from the control generator, coinciding with the first bit of a word, the first word of a group of words, and the first group of a block of groups of words, respectively, and labelled C1, C2 and C3. Simulation of these control signals is accomplished by loading the first element of three global arrays named C1, C2 and C3 with the simulation time of the most recent occurrence of C1, C2 and C3. The "CBITDELAY" primitives are now used to fill out the rest of the arrays with the corresponding delayed signal occurrence time.

When invoked by the occurrence of an event on a data arc, a primitive will compare the present (simulation) time to the time of its control signal. If the times are not identical, the simulator will issue a warning message and commence a diagnostics routine to assist the designer in locating the missing control signal. A weakness of this arrangement is that extra control signals will not be detected by the simulator but would cause erroneous operation in the actual hardware system. This feature should be added in future versions of this simulation.

CONCLUSIONS

The purpose of this research was to investigate the possibility of using ADA as a means of describing digital hardware and performing subsequent simulation of the description. That goal was accomplished with the behavioral description and simulation of a complex-to-magnitude digital signal processing algorithm implementation.

In theory, almost any task that is within the capabilities of one programming language can be accomplished by another language as well, given sufficient time and resources. How well the second language accomplishes the task compared to the first is to a large extent subjective. Programming languages and styles tend to vary in popularity as a matter of personal preference as much as with any true measure of their capabilities (Nash and Saunders 1986). Nevertheless, some additional conclusions were reached as a result of this project:

1. General purpose languages are useful for hardware description. As previously noted, Dietmeyer and Duley questioned the wisdom of allowing hardware description languages to be strongly influenced by general purpose programming languages (Dietmeyer and Duley 1975). It should be noted that this was written prior to the current emphasis on top-down design of digital systems. That same ten years has seen programming languages become ever more powerful and versatile, with an emphasis on strong type checking, readability, maintainability, structuring and standardization that was the exception rather than the rule in 1975. If present hardware description languages are only now evolving to the level of the 1950s programming languages, it seems reasonable to assume that a powerful language like ADA would be far enough advanced to perform adequately in the same role as lesser developed description languages.

2. ADA is not difficult to learn and well worth the effort. It appears that anyone familiar with C or PASCAL can adapt to ADA with little additional effort. Despite concerns of those who argued against full ADA inclusion in VHDL on the grounds that training designers in ADA and VHDL would be too difficult (Nash and Saunders 1986), it is believed that the added power of a full ADA implementation in VHDL would justify any additional training effort.

3. Although not originally intended for the role, ADA is a good choice for hardware description due to its ready availability and support. By 1990, software costs are expected to amount to 90% of all computing costs (Fairley 1986). Organizations without a presently available design automation system might profit more from ADA development with its base of existing software support than from starting from scratch with a special purpose language.

4. Further research is justified. Future work should concentrate on silicon compilation from ADA behavioral descriptions, automatic test vector generation for the simulation, improvement in the simulation tools and additional operators for use by large systems.

REFERENCES

Aylor, J.H.; Waxman, R.; and Scarratt, C. "VHDL - Feature Description and Analysis." IEEE Design and Test of Computers (April 1986): 17-27.

Avres, Ron. "IC Specification Language." In VLSI: The Coming Revolution in Applications and Design. New York: Institute of Electrical and Electronic Engineers, Inc., 1980.

Cohen, Norman H. ADA as a Second Language. New York: McGraw-Hill Book Company, Inc., 1986.

DeMan, H.; Rabaey, P. Six; and Claesen, L. "Cathedral-II: A Silicon Compiler for Digital Signal Processing." IEEE Design and Test of Computers (December 1986): 13-25.

Denyer, Peter, and Renshaw, David. VLSI Signal Processing: A Bit-Serial Approach. Reading, MA: Addison-Wesley Publishing Company, 1985.

Dewey, Allen, and Gadient, Anthony. "VHDL Motivation." IEEE Design and Test of Computers (April 1986): 12-16.

Dietmeyer, Donald L., and Duley, James R. "Register Transfer Languages and Their Translation." In Digital System Design Automation: Languages, Simulation and Data Base. Edited by Melvin A. Brewer. Woodland Hills, CA: Computer Science Press, Inc., 1975.

Downes, V.A., and Bosche, R. Tellaeche. "Discrete Event Modelling in ADA: Implementation and Application." In Proceedings of the Third Joint ADA Europe/ADATEC Conference in Brussels, June 26-28, 1984, pp. 53-63. Edited by J. Teller. Cambridge: Cambridge University Press, 1984.

Fairley, Richard E. Software Engineering Concepts. New York: McGraw-Hill Book Company, Inc., 1985.

Filip, A.E. "A Baker's Dozen Magnitude Approximation and Their Detection Statistics." IEEE Transactions in Aerospace and Electronic Systems AES-12 (1976): 87-89. Quoted in Peter Denyer and David Renshaw, VLSI Signal Processing: A Bit-Serial Approach. Reading, MA: Addison-Wesley Publishing Company, 1985.

Hill, Frederick J., and Peterson, Gerald R. Introduction to Switching Theory and Logical Design, 3rd ed. New York: John Wiley and Sons, Inc., 1984.

Lea, R.M. "VLSI Parallel-Processing Chip Architecture." In Algorithmics for VLSI, pp. 1-32. Edited by C. Trullemans. London: Academic Press, Inc., 1986.

Lipsett, Roger; Marschner, Erich; and Shahdad, Moe. "VHDL - The Language." IEEE Design and Test of Computers (April 1986): 28-41.

Mead, Carver, and Conway, Lynn. Introduction to VLSI Systems. Reading, MA: Addison-Wesley Publishing Company, 1980.

Nash, J.D., and Saunders, L.F. "VHDL Critique." IEEE Design and Test of Computers (April 1986): 54-65.

Price, David. Introduction to ADA. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Rice, Rex, ed. VLSI: The Coming Revolution in Applications and Design. New York: Institute of Electrical and Electronic Engineers, Inc., 1980.

Saib, Sabina. ADA: An Introduction. New York: Holt, Rinehart and Winston, 1985.

United States Department of Defense, American National Standards Institute, Inc. Reference Manual for the ADA Programming Language ANSI/MIL-STD-1815A-1983. New York: Springer-Verlag, 1983.

Wallace, Robert H. Practitioner's Guide to ADA. New York: McGraw-Hill Book Company, Inc., 1986.

Waxman, Ron. "The VHSIC Hardware Description Language - A Glimpse of the Future." IEEE Design and Test of Computers (April 1986): 10-11.

Wegner, Peter. Programming with ADA - An Introduction by Means of Graduated Examples. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1980.

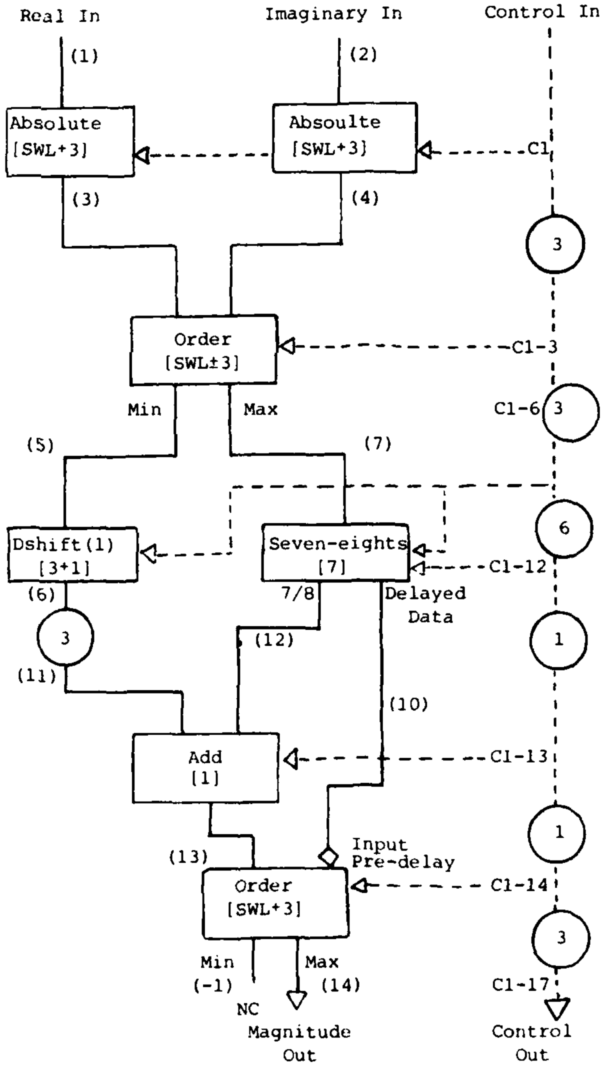


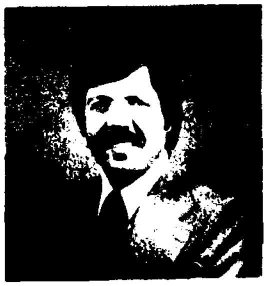
Figure 2. Complex-to-Magnitude Signal Flow Graph.

```
separate(SIMULATE)

procedure SYSTEM is

procedure CONTROL_NETWORK is

begin

   UBITDELAY(c1(0),c1(3),3);
   UBITDELAY(c1(3),c1(6),3);
   UBITDELAY(c1(6),c1(12),6);
   UBITDELAY(c1(12),c1(13),1);
   UBITDELAY(c1(13),c1(14),1);
   UBITDELAY(c1(14),c1(17),3);

end CONTROL_NETWORK;

   procedure COMPLEX_TO MAGNITUDE is

   begin

      case event is

  when 1=>
     ABSOLUTE(input=>1,output=>3,swl=>16,
               ctrl=>c1(0));

  when 2=>
     ABSOLUTE(input=>2,output=>4,swl=>16,
               ctrl=>c1(0));

  when 3:4=>
     ORDER(in1=>3,in2=>4,max=>6,min=>5,
            swl=>16,ctrl=>c1(3));

  when 5=>
     DSHIFT(input=>5,output=>7,power2=>1,
             ctrl=>c1(6));

  when 6..9=>
     SEVEN_EIGHTHS(6,c1(6),c1(12));

  when 10:13=>
     ORDER(in1=>10,in2=>13,max=>14,min=>NC,
            swl=>16,ctrl=>c1(14),del1=>1);

  when 11:12=>
     ADD(addend1=>11,addend2=>12,sum=>13,
          latency=>1,ctrl=>c1(13));

  when others =>
     put("Event = ");
     put(event);
     new_line;
     diagnostics;
     end case;

  end COMPLEX_TO MAGNITUDE;
      begin

         CONTROL:declare
          begin

             CONTROL_GENERATOR(16);
             CONTROL_NETWORK;
          end CONTROL;

DATA:declare

   begin

      KEY_IN(2,c1(0));
      PADIN(1,1,c1(0));
      PADIN(2,2,c1(0));

      case EVENT is
         when 1..13 =>
            COMPLEX_TO_MAGNITUDE;
         when 14 =>
            PADOUT(14,1,c1(17));
            PRINT_OUT(1,c1(17));
         when others =>
            DIAGNOSTICS;
         end case;

   end DATA;

end SYSTEM;
```

Figure 3. System Description of Complex-to-Magnitude

Mark Happel is a senior test engineer with the DMS-10 Division of Northern Telecom Inc., where he is involved with gate-level simulation and functional test programming of ATE equipment. Prior to this, he served as an officer in the U.S. Navy. Happel received his B.S.E.E. from the U.S. Naval Academy in 1980 and his M.S.E. in Electrical Engineering from the University of Central Florida in 1987. His interests include design automation and artificial intelligence. He can be reached at Northern Telecom Inc., NTP Dept. 2634, P.O. Box 13010, Research Triangle Park, NC 27709.

Dr. Brian Petrasko is an associate professor in Electrical Engineering at the University of Central Florida, Orlando, Florida. He received the B.E.E., M.E and D. Eng. (1973), all in electrical engineering, at the University of Detroit. Dr. Petrasko's interests include design automation and digital signal processing.



E-MARS - EMBEDDED MULTI-PROCESSING ADA* RUN-TIME SUPPORT

J. Fuhrer, K. Tupper, M. Levitz, J. Golowner, J. Hetzron

Unisys, Shipboard and Ground Systems Group
M.S. H-3
Great Neck, NY 11020
516-574-3606

ABSTRACT

Previous experience in the development of real-time systems for military applications has demonstrated the need for application support systems to satisfy stringent real-time constraints. This paper presents a software model, under development by the authors, that is to serve as an example of an Ada run-time support system targeted to satisfy rigid performance constraints. As such, it can serve as a basis for:

- Incorporating basic operating system functions into an Ada run-time support system
- Defining a standard Ada compilation interface to this system.

1. INTRODUCTION

The model is identified as the Embedded Multi-Processing Ada Run-Time Support (E-MARS). The purpose of E-MARS is to define an Ada run-time system that can execute with no underlying operating system while supporting up to two independent Ada applications in a dual-CPU shared-memory environment. Due to the lack of an Ada Compiler for E-MARS and its target computer, the following simulations must be supplied:

(1) The interface between E-MARS and the Ada application
(2) The interface between E-MARS and its target computer.

Both these simulations are incorporated within the E-MARS prototype model. As illustrated in Figure 1, E-MARS is currently being developed on a Micro-Vax II Workstation under the auspices of the Vax Ada System.

E-MARS is an Ada Pilot Project that is expected to assist in:

- Identifying potential difficulties in satisfying stringent real-time performance constraints when using Ada
- Determining what must be added to an Ada run-time support system to satisfy the needs of a real-time application
- Defining policies for "implementation dependent" run-time issues
- Evaluating the methodologies and tools employed during the course of Ada program development
- Establishing metrics for Ada program development that is in compliance with DoD-STD-2167.

E-MARS will be developed in compliance with DoD-STD-2167. The E-MARS development process, as illustrated in Figure 2, will evaluate the following set of software development methodologies and tools.

- Structured Analysis techniques with real-time extensions for development of the software requirements
- The Process Abstraction Methodology for Embedded Large Applications (PAMELA) and its supporting tool, AdaGraph, to develop software design requirements.

2.0 E-MARS MODEL

The E-MARS Model consists of the following:

(1) A simulation function **(Perform Simulation Functions)** which performs all processing necessary to create and execute up to two Ada applications based on user-directed inputs
(2) A run-time function **(Perform Run-Time System Functions)** which performs all the processing necessary to support the execution of an Ada application
(3) A support library **(E-MARS Support Library)** which includes predefined program units for data types, time functions, configuration dependent characteristics, input/out, and attributes to support the Ada application simulation.

The context diagram for the E-MARS model is defined in Figure 3. It describes the boundary that separates the E-MARS model from its environment. In addition, it defines the

- - - - -

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office) external interfaces and data flows between the proposed system and its environment. E-MARS is represented as the single transformation called "E-MARS Model".

The E-MARS level 0 data flow diagram (DFD) is defined in Figure 4. It depicts the following two transformations and their interfaces:

(1) Perform Simulation Functions
(2) Perform Run-time System Functions.

The Interface between the E-MARS Support Library and (1) the other components of the E-MARS model, and (2) the host environment is defined in Figure 1.

The Perform Simulation Functions, the Perform Run-Time System Functions, and the E-MARS Support Library are described below.

2.1 PERFORM SIMULATION FUNCTIONS

Perform Simulation Functions is responsible for all processing necessary to create and execute up to two independent Ada applications. An Ada application is created by the user by means of a menu-driven interface. The user specifies the Ada Program structure which includes packages, tasks, procedures, functions, exception handlers, and blocks. Once created, these structures are stored for future access by both this function and Perform Run-Time System Functions. The user then has the option of saving the configuration on a disk file.

The user controls the execution of the application through menu directives. The simulation function interprets these directives, and translates them to corresponding E-MARS Run-Time System Function Calls consisting of either execution commands or simulated interrupts. Perform Run-Time System Functions, after processing the command, will issue an Update Status Request to Perform Simulation Functions. Perform Simulation Functions will then update the user menus to reflect the current execution status of the Ada Program. The process repeats upon receipt of the next user request.

Perform Simulation Functions processing, as illustrated in the Figure 5 data flow diagram, is described below.

2.1.1 GENERATE MENUS

The Generate Menus function provides the user with the capability to create and/or execute models of Ada applications. In response to a user request made via the keyboard, the Generate Menus function creates and displays menus and prompts which lead the user through a logical sequence to create and/or execute the desired Ada applications. Figures 8 through 10 contain samples of these menus.

2.1.2 PROCESS USER INPUT

This function processes the user's inputs which are generated at the keyboard. User input data consists of either a start command, configuration data, execution control data or simulation commands. The user responds to displayed menus which require the user to select from a list of options or to enter specified data. The Process User Input function determines if the user's input is valid for the menu currently displayed. If valid, the input data is passed to the appropriate function (Process Configuration Control, Process Execution Control, or Process Data Storage Control) for further processing. If the input data is invalid, an error message is issued.

2.1.3 PROCESS CONFIGURATION CONTROL

The Process Configuration Control function is responsible for creating Ada program templates from user inputs and storing corresponding data structures for future execution. This function is also responsible for incorporating packages included in a "with" clause for the structure being created.

2.1.4 UPDATE STATUS

The Update Status function is activated upon receipt of an Update Status Request from Perform Real-Time System Functions. The Update Status Request is issued upon completion of an event including:

- Program unit elaboration
- Conclusion of exception handling processing
- Entry, exit, and suspension of a program frame.

This function updates a Status Window on the screen, describing the current state of each program unit.

2.1.5 PROCESS EXECUTION CONTROL

Based on dynamic user directives, the Process Execution Control function formats and issues calls to Perform Run-Time System Functions. Capability is provided for the user to issue commands to control elaboration, interrupt generation, task execution including rendezvous, entering and exiting from blocks, and subprogram calls and returns.

2.1.6 PROCESS DATA STORAGE CONTROL

This function interfaces with a disk file to save and retrieve application configuration data.

2.2 PERFORM RUN-TIME SYSTEM FUNCTIONS

Perform Run-Time System Functions is responsible for performing both basic operating system functions and those functions necessary to support the Ada language as defined in ANSI/MIL-

STD-1815A-1983, but which are characteristic of operating system functions. Perform Run-Time System Functions processing, as illustrated in the Figure 6 data flow diagram, is described below.

2.2.1 PERFORM E-MARS START-UP

This function performs elaboration of E-MARS Support Library Packages, and initiates both the time management and time-slicing functions.

2.2.2 ELABORATE PROGRAM UNIT

This function elaborates declarative items within application program unit structures. Elaboration is the process by which a declaration achieves its effect. The E-MARS Model will create an associated control block for the particular declarative object and allocate the required storage. During this process, a list of tasks that are to be activated is created. The Frame Management Function will initiate the activation elaboration. If an exception is raised during elaboration, the execution is abandoned, and the processing is as defined in Process Exception. When the last declaration within a declarative part is elaborated, a "Frame Body Entry Command" is issued to the Frame Management Function.

2.2.3 PROCESS EXCEPTION

This function supports and provides identical response in accordance to ANSI/MIL-STD-1815A-1983 for both predefined and application defined exceptions. The predefined exceptions supported by the E-MARS Model include:

CONSTRAINT_ERROR
PROGRAM_ERROR
NUMERIC_ERROR
STORAGE_ERROR
TASKING_ERROR

The Process Exception function accepts an "Exception Command" generated by Perform Simulation Functions, and transfers control to the corresponding exception handler. The data structures associating program frames and exception handling are considered to be a compiler interface function. As such, application program data structures for exception handling, which were created by the user during Perform Simulation Functions, are now utilized.

The E-MARS response depends upon the state of the frame in which the exception was raised and whether or not a corresponding handler is defined for that frame. Once the corresponding exception handler is chosen and action taken in accordance with ANSI/MIL-STD-1815, the Process Exception function will appropriately update the status of all affected program units.

2.2.4 TASK MANAGEMENT

The Task Management function performs all task-related support functions. These functions refer to controlling tasks across two processors and the processing of task execution statements such as select statements, entry calls, accept statements, etc. Task Management processing, as illustrated in the Figure 7 data flow diagram, is described below.

Process Entry Call — Evaluates the called task environment to determine if a rendezvous is possible. Processing functions are provided to support simple, timed and conditioned entry calls, and to perform the rendezvous.

Rendezvous End — Restores the proper execution of the called task, if necessary, and re-schedules the caller. If a Rendezvous End occurs with a pending exception, Process Exception is invoked to propagate the exception.

Task Control — Performs all functions related to time-slicing scheduling and context switching.

Process Abort Statement — Performs actions corresponding to the state of the task to be aborted. The task will become "completed" if suspended at an accept, select or delay statement. If not completed, the task is designated "abnormal".

Process Accept Satement — If there are no callers, the state of the task is changed to "waiting at an accept". If the entry was called, the caller will be removed from the entry queue and the accept statement executed at the priority of the higher of the two tasks in the rendezvous.

Process Selective Wait — Considers all open accept statements within the select statement. If one or more exists, one is arbitarily selected. If no rendezvous is immediately possible, and an open else part exists, it is executed. If no immediate rendezvous is possible and a delay statement exists, the state of the task is changed to "suspended at a selective wait" and delay processing is executed. The open terminate will be selected if all terminate conditions are met.

Process Delay Satement Performs delay processing and updates the task status to "suspended at a delay statement".

Terminate a Task Determines if all dependent tasks have terminated or are waiting on an open terminate alternative. If these conditions are true, the state of the task is changed to "terminated", and a request is made to the Memory Management function to deallocate storage associated with this task. Otherwise, the state of the task is changed to "waiting for termination".

2.2.5 INTERRUPT MANAGEMENT

The Interrupt Management function serves as an intermediary between asynchronous hardware interrupts and various other parts of the Ada run-time support environment. Interrupts are actually traps which signal conditions that are handled immediately and are transparent to the Ada System or signal conditions that require extended processing by other parts of the system.

The target computer hardware interrupt generation is simulated by Perform Simulation Functions. In response to a user request, *Perform Simulation Functions generates an* E-MARS Run-Time System Function Call denoting the type of interrupt to be handled. Perform Run-Time System Function's Interrupt Management function is then invoked to process the interrupt. Interrupts which correspond to predefined Ada exceptions are passed to the Task Management function. The interrupt corresponding to the conclusion of a timing interval is passed to the Process Timer Request function. Other interrupts such as those indicating conclusion of an I/O request, hardware fault or interprocessor communication are passed to associated processing functions.

2.2.6 FRAME MANAGEMENT

This function provides support for entering and leaving a new program scope. Upon entry into a frame, Frame Management will place a corresponding Frame ID on the Frame Execution Stack, and issue an Elaboration Request for the declarations within the frame. Upon exit, the Frame ID is removed from the stack. If an exception was raised during execution of the frame, the Process Exception function is invoked to process the exception.

2.2.7 MEMORY MANAGEMENT

This function allocates and deallocates storage at run time.

2.2.8 TIME MANAGEMENT

This function monitors, synchronizes, and adjusts the accuracy of the Real Time Clocks.

2.3 E-MARS SUPPORT LIBRARY

The E-MARS Support Library packages include:

Package Standard
Package System
Package Calendar
Package Text_IO
Package Low_level_IO
Package Attribute.

The implementation dependent features of these packages are defined for the target computer. The Host Ada Support Library packages will be used to support execution of the E-MARS Mode.

2.3.1 PACKAGE STANDARD

Package STANDARD is normally provided as an integral part of the Ada Compiler implementation. Package STANDARD contains declarations of those identifiers predefined in the language and their *applicable operations.* For this simulation, the data types and their operations as described in Appendix C of ANSI/MIL-STD-18115A-1983 are provided.

In this context, type DURATION is defined as:

```
   type DURATION is delta 0.001 range (-86400.0
.. 86400.0);
```

2.3.2 PACKAGE SYSTEM

Package System contains implementation dependent characteristics. For this simulation a sample of some of the parameters included in the visible portion of the specification is as follows:

```
type ADDRESS is integer range (0..524288);
type NAME is (E-MARS, E-MARS_SINGLE, E-MARS_
DUAL);

SYSTEM_NAME    :constant NAME    := E-MARS;
MEMORY_SIZE    :constant         := 524288;   --
512K locations
MIN_INT        :constant         := 2E31;
MAX_INT        :constant         := 2E31 -1;
MAX_DIGITS     :constant         := 15;
MAX_MANTISSA   :constant         := 16;
FINE_DELTA     :constant         :2.0E - 31;
TICK           :constant         :10.0E -2;

subtype PRIORITY is integer range (1..10);
```

2.3.3 PACKAGE CALENDAR

All procedures and functions as described in paragraph 9.6 of ANSI/MIL-STD-1815A-1983 will be provided. Each of these procedures and functions will provide identical parameters when accessed concurrently by each CPU.

2.3.4 PACKAGE TEXT_IO

The E-MARS model makes direct use of the underlying Ada support system for implementation of this package.

2.3.5 PACKAGE LOW_LEVEL_IO

The E-MARS model will provide SEND_CONTROL and RECEIVE__CONTROL procedures for a single fictitious device.

2.3.6 PACKAGE ATTRIBUTE

This package contains both language-defined attributes that are supported by E-MARS and E-MARS-defined attributes. Attributes are basic characteristics of Ada data types and data objects. An attribute denotes a basic operation such as a function, type or range. The predefined language attributes, as defined in ANSI/MIL-STD-1815A-1983, that are supported by E-MARS are as follows:

P'ADDRESS
P'CALLABLE
P'COUNT
P'TERMINATED.

The E-MARS attributes defined for tasks are as follows:

P'SCHEDULED Yields the value false when the task P is not in a scheduling queue (it is either completed, terminated, in a delay or running)

P'RUNNING Yields the value false if the task P is not the Current Task in either CPU

P'DELAYED Yields the value false if task P is not in a time delay.

3.0 STATUS OF THE PROJECT

The requirements for the E-MARS Model have been generated using Structured Analysis with real-time extensions. The Software Requirements Specification has been completed and is under review. This document is in compliance with DoD-STD-2167.

SUMMARY

This paper describes an Ada Pilot Project, the objectives of which are to gain expertise in both Ada Technology and complementary development methodologies, to understand their limitations, and to provide solutions for these limitations.

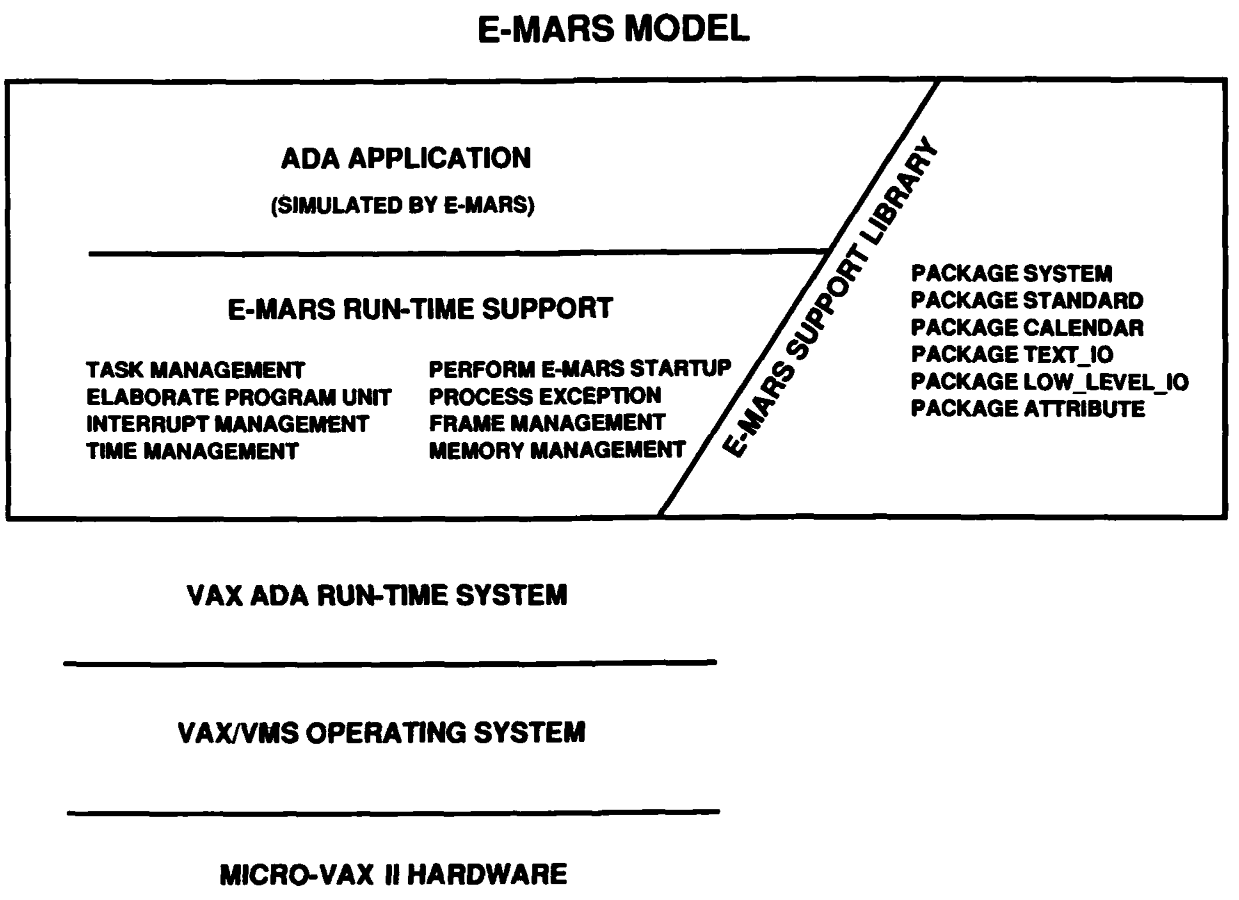
REFERENCES

[1] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983

[2] S. Mellor and P. Ward, "Structured Development for Real-Time Systems", Yourdon Press Computing Series, 1986

[3] G. Cherry, "Pamela Designers Handbook", Analytical Sciences Corporation, 1986

[4] Department of Defense, "DoD-STD-2167", June 1985



E-MARS MODEL

ADA APPLICATION

(SIMULATED BY E-MARS)

E-MARS RUN-TIME SUPPORT

TASK MANAGEMENT
ELABORATE PROGRAM UNIT
INTERRUPT MANAGEMENT
TIME MANAGEMENT

PERFORM E-MARS STARTUP
PROCESS EXCEPTION
FRAME MANAGEMENT
MEMORY MANAGEMENT

E-MARS SUPPORT LIBRARY

PACKAGE SYSTEM
PACKAGE STANDARD
PACKAGE CALENDAR
PACKAGE TEXT_IO
PACKAGE LOW_LEVEL_IO
PACKAGE ATTRIBUTE

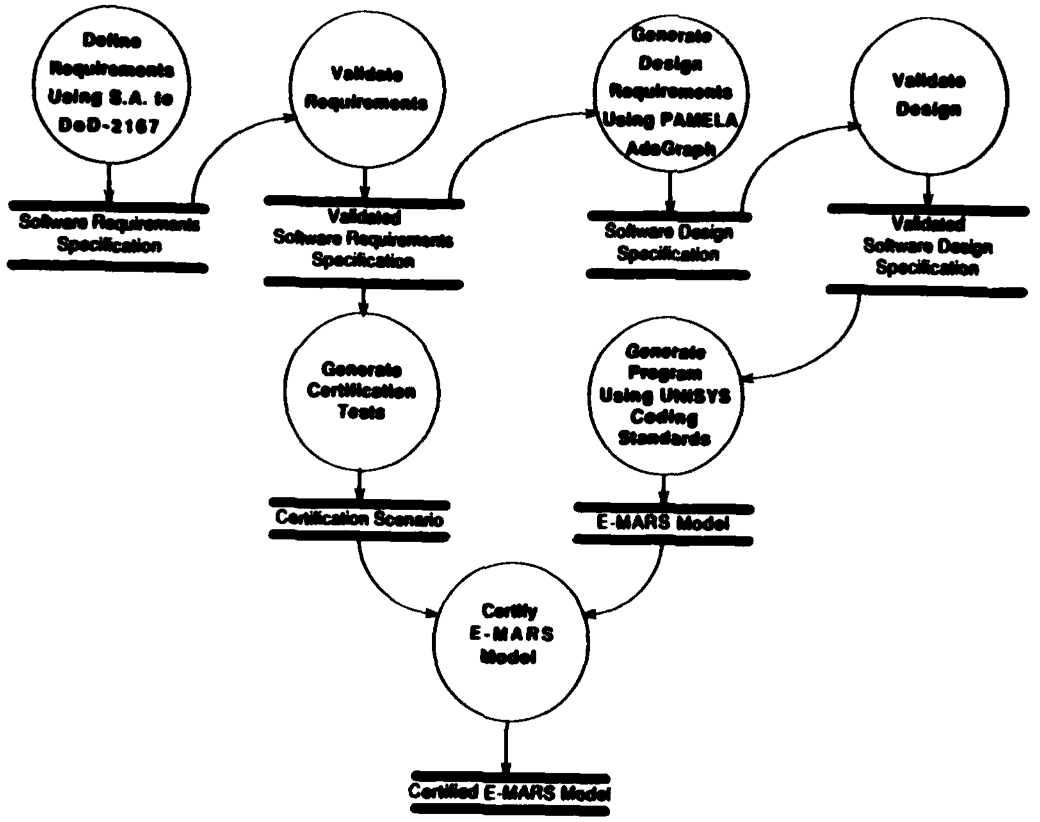
VAX ADA RUN-TIME SYSTEM

VAX/VMS OPERATING SYSTEM

MICRO-VAX II HARDWARE

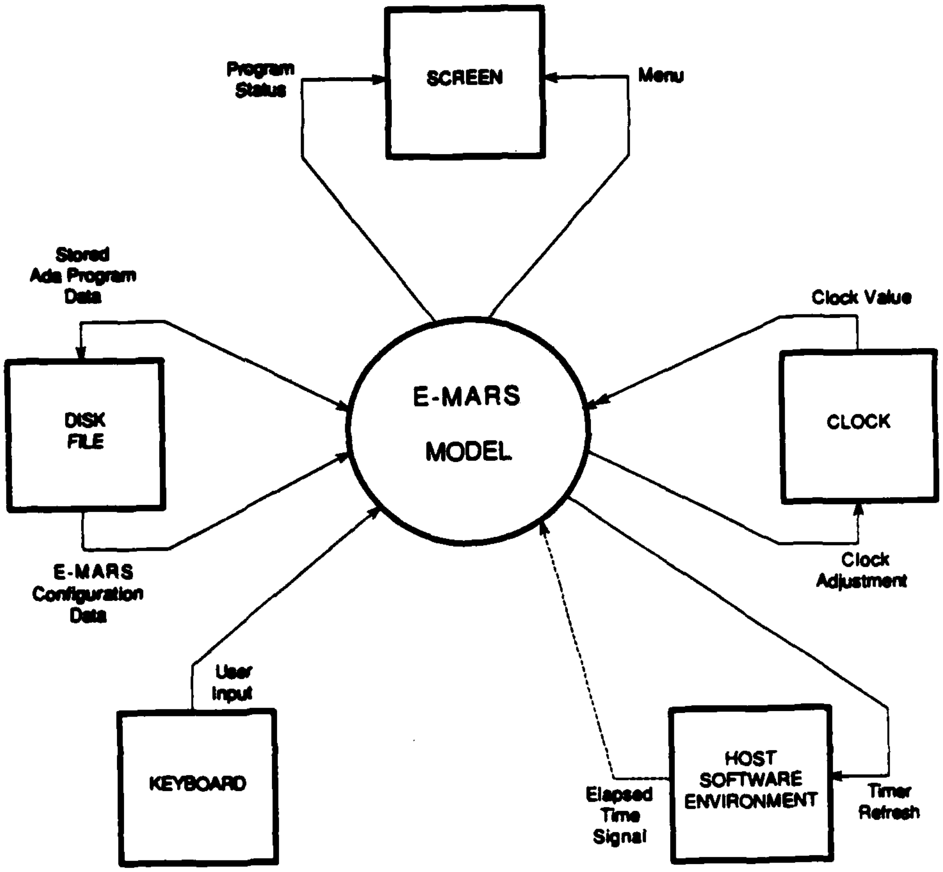
E-MARS Model Development Environment

Figure 1


Define Requirements Using S.A. to DoD-2167
Software Requirements Specification
Validate Requirements
Validated Software Requirements Specification
Generate Design Requirements Using PAMELA AdaGraph
Software Design Specification
Validate Design
Validated Software Design Specification
Generate Certification Tests
Certification Scenario
Generate Program Using UNISYS Coding Standards
E-MARS Model
Certify E-MARS Model
Certified E-MARS Model

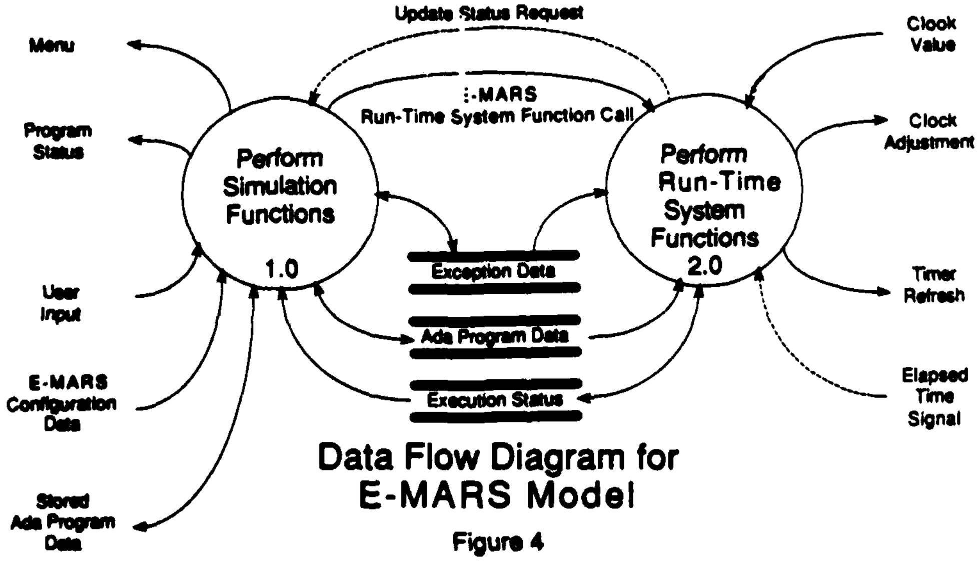
E-MARS Development Process

Figure 2



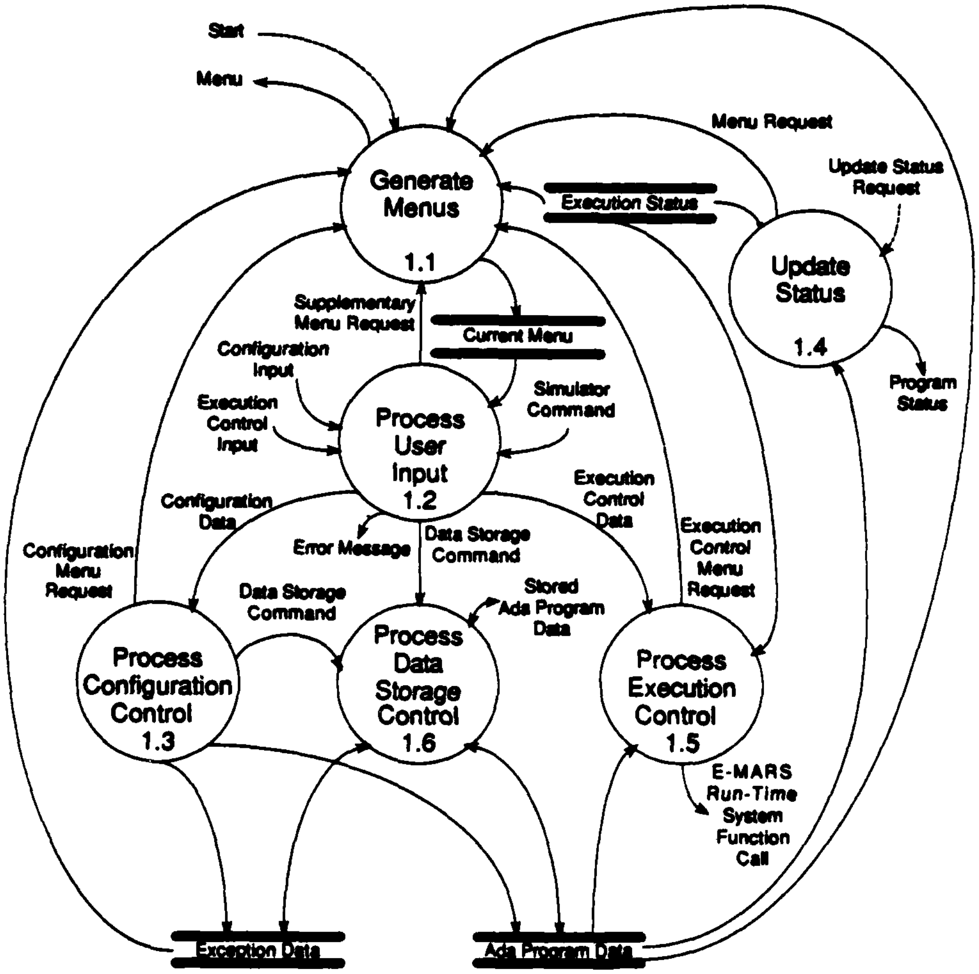
Context Diagram for E-MARS Model

Figure 3

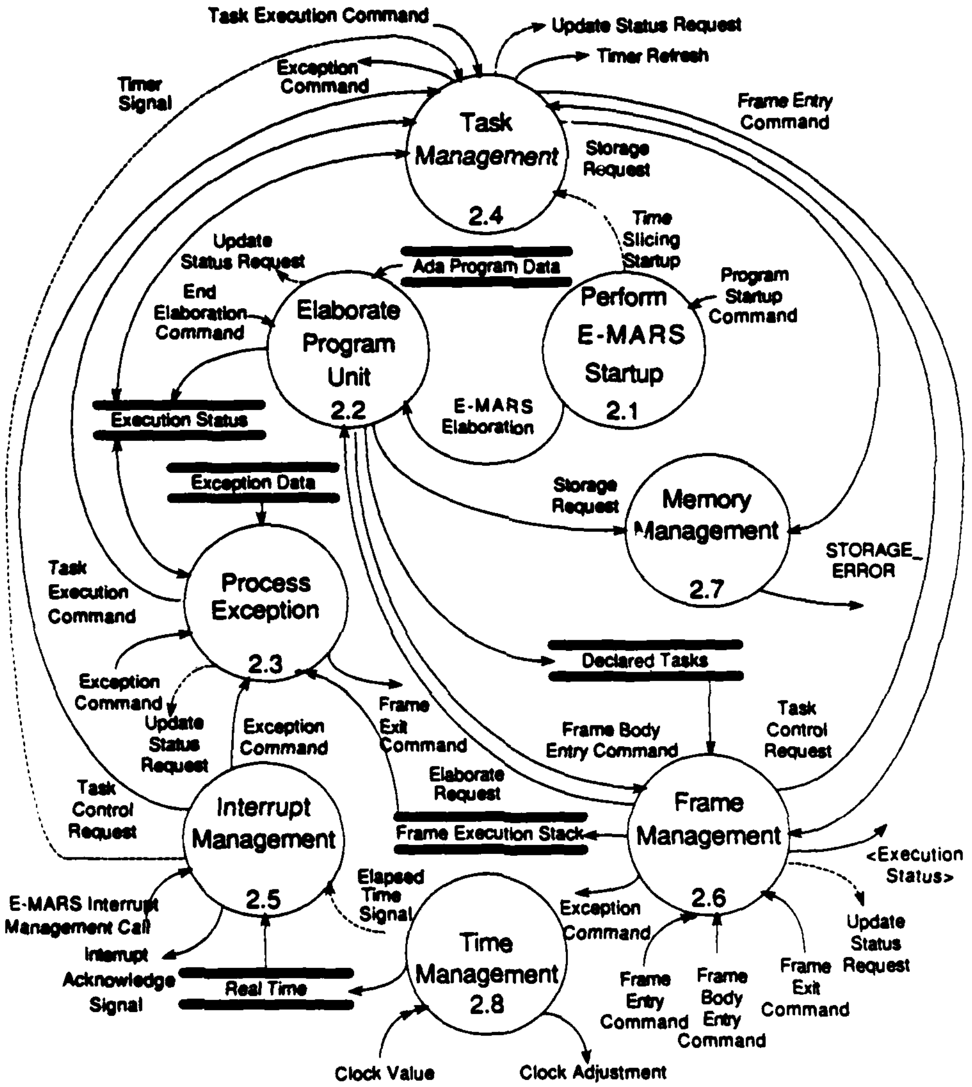


Data Flow Diagram for E-MARS Model

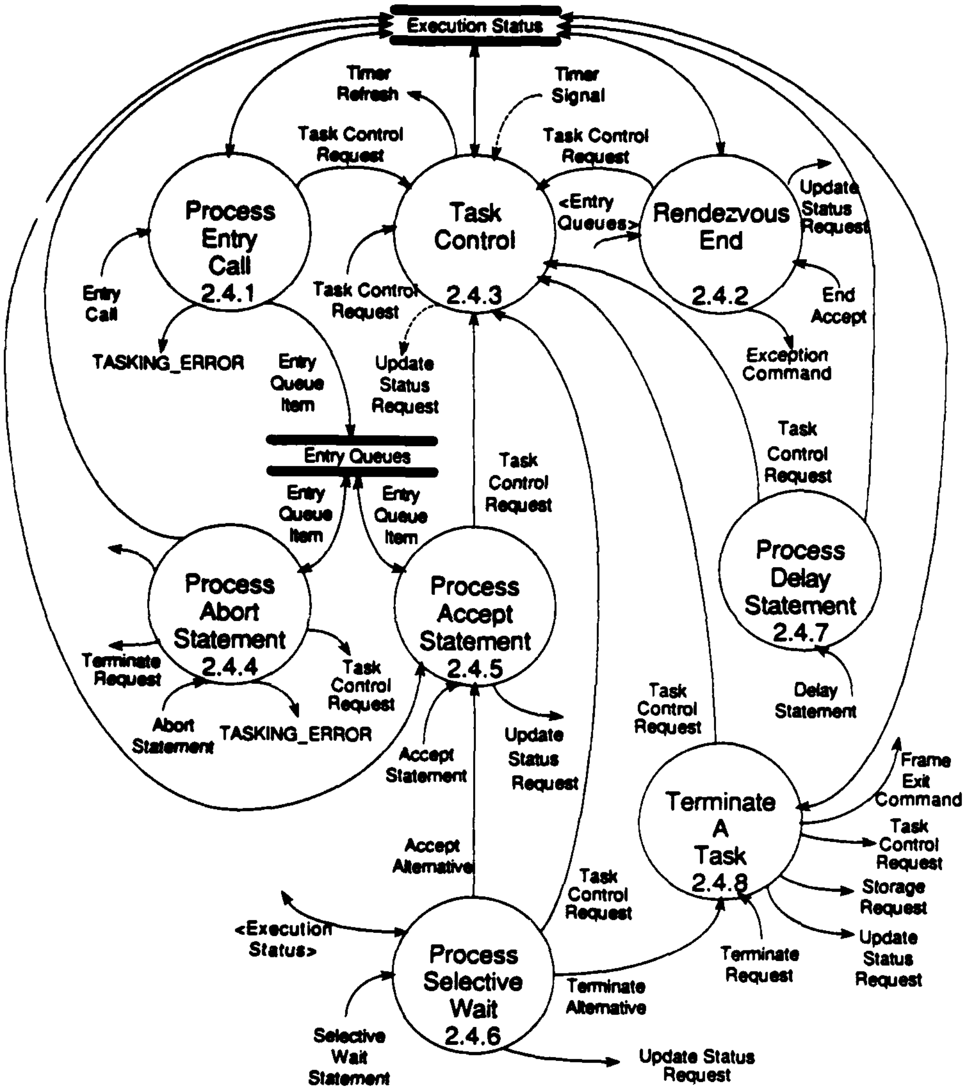
Figure 4



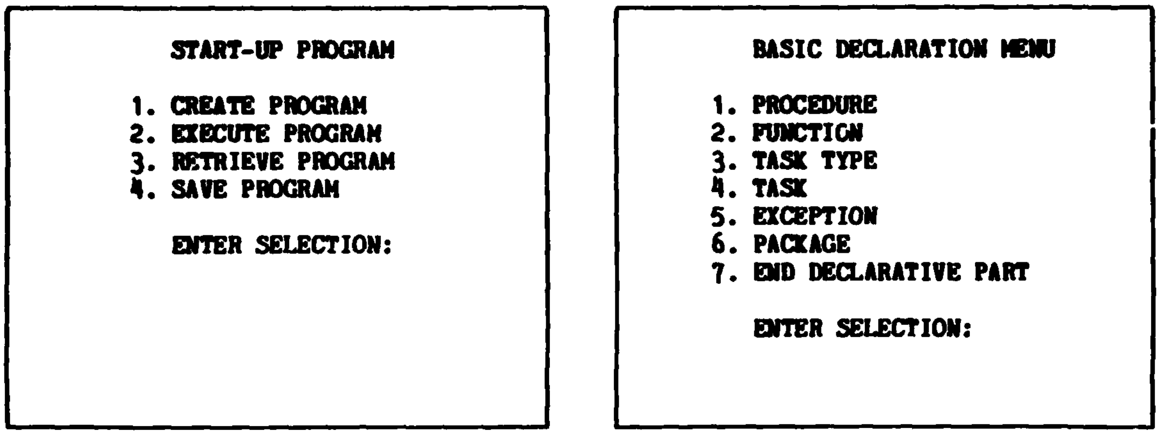
Data Flow Diagram for
E-MARS Model
Perform Simulation Functions 1.0
Figure 5



Data Flow Diagram for
E-MARS Model
Perform Run-Time System Functions 2.0
Figure 6



Data Flow Diagram for
E-MARS Model
Task Management 2.4
Figure 7

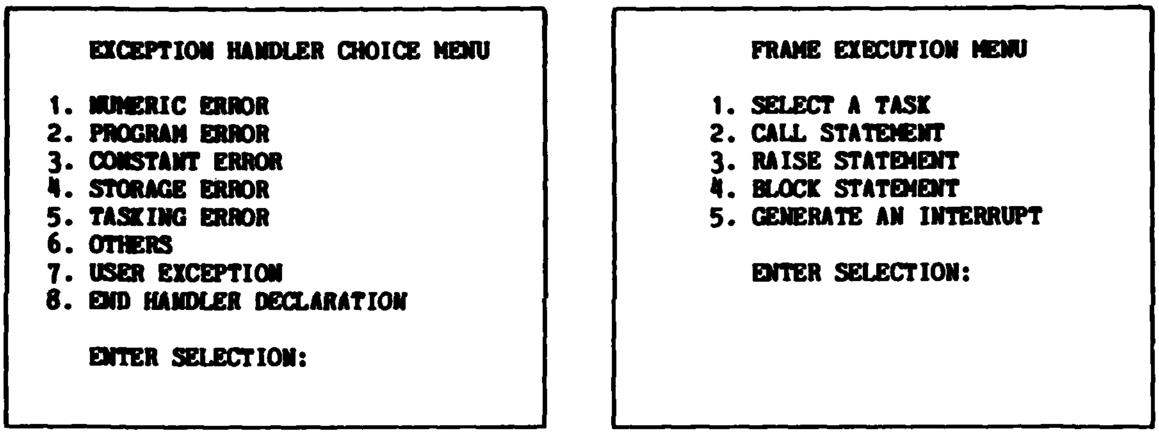


```
START-UP PROGRAM

1. CREATE PROGRAM
2. EXECUTE PROGRAM
3. RETRIEVE PROGRAM
4. SAVE PROGRAM

   ENTER SELECTION:
```

```
BASIC DECLARATION MENU

1. PROCEDURE
2. FUNCTION
3. TASK TYPE
4. TASK
5. EXCEPTION
6. PACKAGE
7. END DECLARATIVE PART

   ENTER SELECTION:
```

Sample Menus

Figure 8

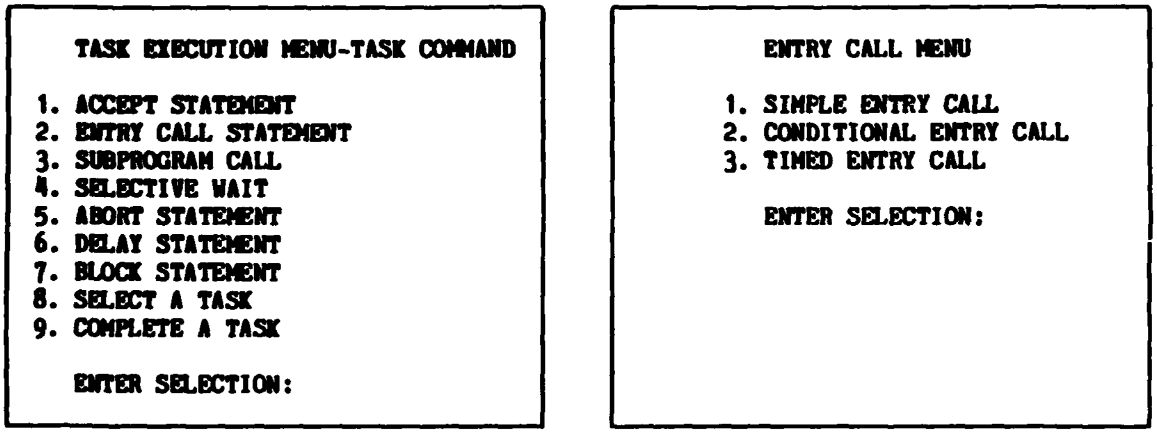


```
EXCEPTION HANDLER CHOICE MENU

1. NUMERIC ERROR
2. PROGRAM ERROR
3. CONSTANT ERROR
4. STORAGE ERROR
5. TASKING ERROR
6. OTHERS
7. USER EXCEPTION
8. END HANDLER DECLARATION

   ENTER SELECTION:
```

```
FRAME EXECUTION MENU

1. SELECT A TASK
2. CALL STATEMENT
3. RAISE STATEMENT
4. BLOCK STATEMENT
5. GENERATE AN INTERRUPT

   ENTER SELECTION:
```

Sample Menus

Figure 9



```
TASK EXECUTION MENU-TASK COMMAND

1. ACCEPT STATEMENT
2. ENTRY CALL STATEMENT
3. SUBPROGRAM CALL
4. SELECTIVE WAIT
5. ABORT STATEMENT
6. DELAY STATEMENT
7. BLOCK STATEMENT
8. SELECT A TASK
9. COMPLETE A TASK

   ENTER SELECTION:
```

```
ENTRY CALL MENU

1. SIMPLE ENTRY CALL
2. CONDITIONAL ENTRY CALL
3. TIMED ENTRY CALL

   ENTER SELECTION:
```

Sample Menus

Figure 10

BIOGRAPHICAL SKETCHES

Mr. Fuhrer has 17 years' experience in development and management of real-time application and simulation software for shipboard, control, navigation, fire control and weapon systems, and related trainer systems. In his current assignment as Research Department Head for Software Technology, he is responsible for the development of an integrated toolset. The toolset is intended to improve productivity of software and systems engineers by providing "intelligent" automation support for software life-cycle activities from requirement specification through software module testing, and providing language and methodology support for HOLs such as Ada, CMS-2, Pascal, and Fortran. Previously, Mr. Fuhrer worked eight years in microwave tube research and development and six years in radar and weapon systems development.

Mr. Fuhrer received a BSEE from Princeton University in 1954, and MSEE from Columbia University in 1960, and an MS in Management Engineering from Long Island University in 1972. He received an MS in Computer Science from Polytechnic University in 1986, and is currently matriculated in the doctoral program. Mr. Fuhrer served two years of active duty (1954 - 1956) as a Naval officer.

Mr. Tupper has worked in real-time Command and Control Systems development for 15 years. He is Engineering Section Head for development of software life-cycle support tools and Ada technology. Previously, Mr. Tupper worked on several real-time simulators and command/control systems. The most recent was the Spanish Navy Program where he was the senior engineer responsible for design and development of Simulation Support Processor software for the land-based testing of FFG- and CV-class naval ships.

Mr. Tupper received his bachelor's degree in 1973 and his master's degree in 1975 (in Computer Science) from Pratt Institute. Mr. Tupper has published several IR&D reports and papers on operating systems development.

Ms. Levitz has 20 years of in-depth experience in the design, documentation, implementation, and test (including quality control) of software systems and software support. Her primary emphasis has been in development of embedded real-time operating systems for military applications. She was responsible for design and development of the Memory Processor Operating System (MPOS) on the Trident II navigational subsystem shipboard computers. She originated the concept of a System Generation Program and a casualty dump embedded in MPOS to facilitate correction of programming problems. Ms. Levitz is group leader of an Ada Core Group whose responsibilities include the development of E-MARS.

Ms. Levitz is the author of five operating system-related publications. She earned a bachelor's degree in chemistry and mathematics from the University of Connecticut in 1955, a bachelor's degree in mechanical engineering from the University of Hartford in 1957, and a master's degree in mechanical engineering and applied mathematics from Rensselaer Polytechnic Institute in 1962. In June 1987, Ms. Levitz was selected as a winner of the Unisys Excel award for her work on the Trident II navigational subsystem operating system.

Mr. Golowner has eight years of software engineering experience, including five years at Unisys. Mr. Golowner designed, implemented, and tested major program components for Command, Control, and Communication Applications including those for the SEALITE, HELF, and TAC-

DEW projects. Mr. Golowner has specialized in CAMAC systems integration software. He designed and imp'emented software functions for a Memory Processor Operating System (MPOS) used in the TRIDENT II Navigational Subsystem. Mr. Golowner is a member of the Unisys Ada Core Group, involved in the development of E-MARS. He is also involved in the definition and development of the validation function of an integrated software engineering environment supporting real-time structured analysis, structured design, automated module test, automated analysis, and knowledge-based design synthesis. He is also responsible for all software functions in the Planes Control Computer of the Large Scale Vehicle.

Mr. Golowner re‿eived his BS in Computer Science and Applied Mathematics from the State University of New York at Albany in 1980, and his MS in Computer Science from Polytechnic Institute of New York in 1983.

Ms. Hetzron has seven years of software engineering experience with five years at Unisys. Ms. Hetzron designed, implemented, and tested major program components for Command, Control, and Communication applications including those for the SEALITE, HELF, and TAC-DEW projects. She designed and implemented software functions for a Memory Processor Operating System (MPOS) used in the TRIDENT II Navigational Subsystem. As a member of the Unisys Ada Core Group, she is involved in the development of E-MARS. She is also involved in the development of the data-flow diagram graphical editor of an integrated software engineering environment supporting real-time structured analysis, structured design, automated module test, automated analysis, and knowledge-based design synthesis.

Ms. Hetzron received her BA in Computer Science from Queens College in 1980, and her MS in Computer Science from Polytechnic Institute of New York in 1983.

AN ADA IMPLEMENTATION OF AN ITERATOR FOR QUADCODES

Larry Latour
Assistant Professor of Computer Science

University of Maine
Orono, ME 04469

Abstract

An iterator is a useful mechanism for traversing through an abstract collection of objects. In some cases the implementation of the iterator is straightforward, in some cases not. In this paper one such case of the latter, an iterator for quadcodes, is considered. Such codes are useful for representing two-dimensional objects in a plane, and have applications in computer graphics and image processing. An iterator implementation for a binary tree structure using Ada tasks is first presented, and the general implementation design is then applied to the implementation of a quadcode iterator. In addition, problems arising due to the asymmetric nature of the rendezvous and syntactical restrictions on the placement of accept statements are discussed.

I. Introduction

The iteration abstraction, or iterator, is a useful mechanism for traversing through an abstract collection of objects, and has been widely discussed in the literature [1,4,11,12]. It typically consists of two operations FIRST and NEXT, with FIRST returning the "first" object in the collection and repeated calls to NEXT returning subsequent objects in the collection. One use of an iterator might be to search for the existence of a particular element in a set. Iterators for ordered sets, stacks, queues, trees, etc., are also common. For a wide range of examples, see Booch [2].

In Ada, the structure of an abstract collection of objects is typically encapsulated in a package body and the iterator in the form of its FIRST/NEXT operations is available to the user in the package specification. In some cases the implementation of the iterator is straightforward, in some cases not. In this paper one such example of the latter, an iterator for quadcodes, is considered. Such codes are useful for representing two-dimensional objects in a plane, and have applications in computer graphics and image processing. Shu-Xiang and Loew describe the properties of quadcodes in two related papers [7,8] and Samet presents a thorough survey of a related approach, quadtrees [10]. An implementation of such a quadcode iterator is being used as part of the object storage and retrieval subsystem of SeeGraph, a graphical tool developed in the University of Maine Computer Science Advanced Projects Lab [5,6].

The paper is organized as follows. Two alternate implementations of a simple symbol table are first presented and discussed, one using a singly linked list and the other using a binary tree structure. After an iterator implementation for the binary tree structure is presented using Ada tasks, the general implementation design is then applied to a quadcode iterator. In the process of implementing these iterators a number of interesting problems arising due to the asymmetric nature of the rendezvous and syntactical restrictions on the placement of accept statements are discussed.

II. Two Alternate Implementations of A Symbol Table Iterator

Consider the implementation of an iterator for a symbol table, i.e., a table consisting of a collection of (symbol,information) pairs with store and retrieve operations. We assume that the iterator part of the specification is represented by two operations: GET_FIRST(SYMBOL,INFO) and GET_NEXT(SYMBOL,INFO).

If the symbol table is implemented using a singly linked list of pairs, the current state of the iterator is simply an access object to the current pair in the list. This "state" pointer is then updated as a side-effect of each call to GET_FIRST/GET_NEXT (See figure 1).

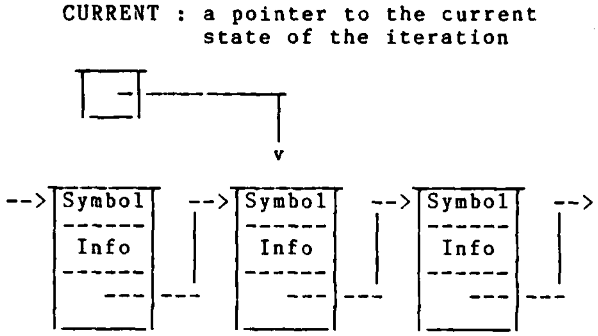


figure 1

Consider an alternate symbol table implementation, one using a binary tree structure in such a way that an inorder traversal of the tree yields the list of pairs in alphabetical order by SYMBOL. In order to implement the GET_FIRST/GET_NEXT operations, we must manage and save the state of the inorder tree walk between calls to the iterator operations. As an example, consider the symbol table of program variable names in figure 2.

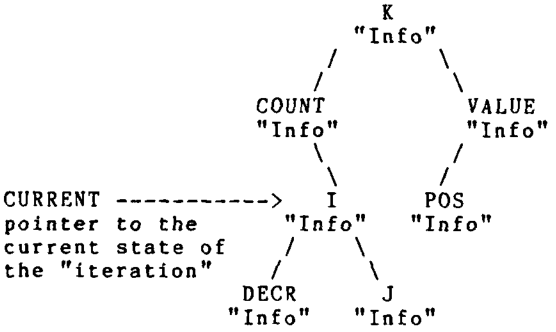


figure 2

Since CURRENT points to the tree node containing the symbol I, a subsequent call to GET_NEXT will update CURRENT to point to the tree node containing J.

An interesting implementation for this iterator comes to mind when we consider how an inorder traversal of a binary tree is typically implemented, i.e., recursively. We can implement the iterator operations with an Ada task that performs a recursive inorder traversal of the symbol table, rendezvousing with the user program through a GET_FIRST/GET_NEXT procedural interface. The user process and the iterator task act as co-routines, with the iterator task incrementally executing as each pair is requested. The state of the iterator is embedded within the implicit run-time stack of the iterator task. The details of this implementation are presented in the following paragraphs.

As suggested, a partial specification of the symbol table looks like:

```
package SYMBOL_TABLE is

   type SYMBOLTYPE is STRING;
   type INFOTYPE is STRING;
      .
      .
-- Iterator operations and exceptions

procedure GET_FIRST(SYM: out SYMBOLTYPE;
                    INFO: out INFOTYPE);
procedure GET_NEXT(SYM: out SYMBOLTYPE;
                   INFO: out INFOTYPE);

   END_OF_ITERATION,
   UNINITIALIZED_ITERATOR: exception;

end SYMBOL_TABLE;
```

The data structure of the symbol table is a tree of dynamically allocated nodes, defined below:

```
package body SYMBOL_TABLE is
   type NODEINFO;
   type NODE is access NODEINFO;
   type NODEINFO is record
      SYM: SYMBOLTYPE;
      INFO: INFOTYPE;
      LEFT,RIGHT: NODE;
   end record;

   ROOT: NODE;
     .
     .
```

As suggested, the iterator is implemented using a task to manage the current state of the iterator. The task and the user program act much like "co-routines", with the additional feature that the task can "look ahead" and retrieve the next element in the iteration while the user program is processing the current element. We first present the Buhr diagrams [3] in figure 3, and follow these with Ada implementations for both the task specification and the procedures GET_FIRST and GET_NEXT encapsulating the task access protocol.

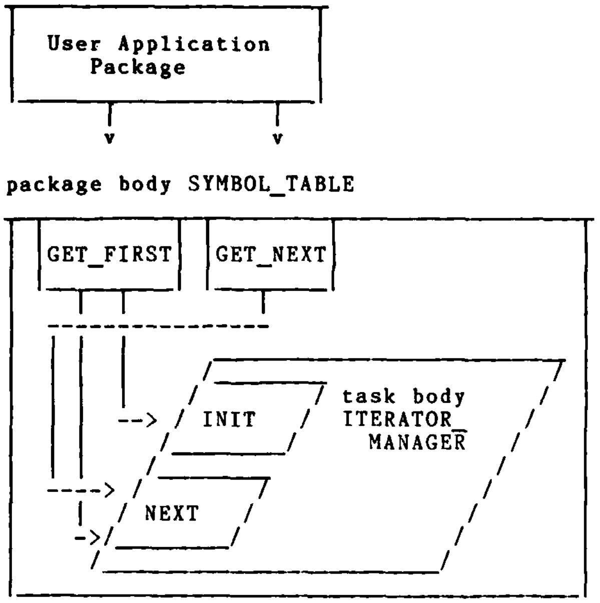


figure 3

```
task ITERATOR_MANAGER is
   entry INIT;
   entry NEXT(S: out SYMBOLTYPE;
              I: out INFOTYPE;
              VALID_NEXT: out BOOLEAN;
              END_OF_ITER: out BOOLEAN);
end ITERATOR_MANAGER;

procedure GET_FIRST(SYM: out SYMBOLTYPE;
                    INFO: out INFOTYPE) is
   S: SYMBOLTYPE;
   I: INFOTYPE;
   VALID: BOOLEAN;
   EOF: BOOLEAN;
begin
   ITERATOR_MANAGER.INIT;
   ITERATOR_MANAGER.NEXT(S,I,VALID,EOF);
   if EOF then
      raise END_OF_ITERATION;
   end if;
   SYM:= S;
   INFO:= I;
end GET_FIRST;


procedure GET_NEXT(SYM: out SYMBOLTYPE;
                   INFO: out INFOTYPE) is
   S: SYMBOLTYPE;
   I: INFOTYPE;
   VALID: BOOLEAN;
   EOF: BOOLEAN;
begin
   ITERATOR_MANAGER.NEXT(S,I,VALID,EOF);
   if not VALID then
      raise UNINITIALIZED_ITERATOR;
   end if;
   if EOF then
      raise END_OF_ITERATION;
   end if;
   SYM:= S;
   INFO:= I;
end GET_NEXT;
```

We now consider the implementation of the ITERATOR_MANAGER task. To aid us in understanding this implementation, we first view the abstract execution of this task as a finite state machine (FSM) pictured in figure 4.

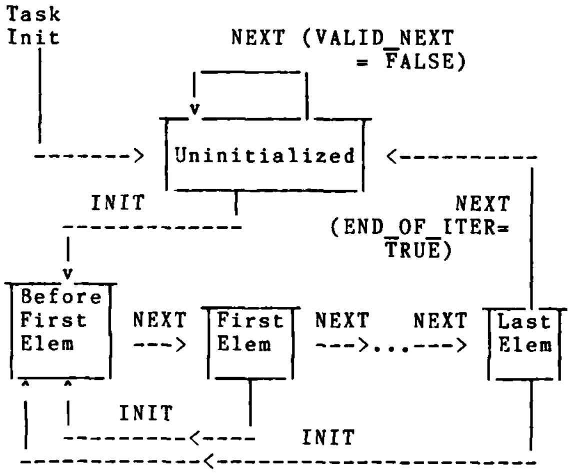


figure 4

The diagram in figure 4 describes the following behavior:

1. The INIT operation is always legal and will always put the FSM into the state "Before First Elem".

2. The behavior of the NEXT operation will vary depending on the state of the FSM: it will normally cause a transition to the next element in the iteration, except when either there is no next element or the iteration has not been initialized with an INIT.

Our first attempt at an implementation of the ITERATOR_MANAGER is presented next. Notice that the task utilizes an embedded recursive routine ITERATE, which will perform the required tree traversal and rendezvousing.

```
task body ITERATOR_MANAGER is
   NORMAL: BOOLEAN:= TRUE;
   INITIALIZED: BOOLEAN:= FALSE;
   BREAKOUT: exception;
   procedure ITERATE(N: in NODE) is
   begin
      if N /= null then
         ITERATE(N.LEFT);

         select
           accept INIT do
              NORMAL:= FALSE;
           end INIT;
         or
           accept NEXT(S: out SYMBOLTYPE;
              I: out INFOTYPE;
              VALID_NEXT: out BOOLEAN;
              END_OF_ITER: out BOOLEAN;
                 S:= N.SYMBOL;
                 I:= N.INFO;
                 VALID_NEXT:= TRUE;
                 END_OF_ITER:= FALSE;
           end NEXT;
         end select;

         if not NORMAL then
            raise BREAKOUT;
         end if;

         ITERATE(N.RIGHT);
      end if;
   end ITERATE;

begin -- BODY OF ITERATOR_MANAGER
   loop
      while not INITIALIZED loop
         select
           accept INIT do
              INITIALIZED:= TRUE;
           end INIT;
         or
           accept NEXT(S: out SYMBOLTYPE;
              I: out INFOTYPE;
              VALID_NEXT: out BOOLEAN;
              END_OF_ITER: out BOOLEAN) do
                 VALID_NEXT:= FALSE;
           end NEXT;
         end select;
      end loop;
      begin
         ITERATE(ROOT);
      exception
         when BREAKOUT => null;
      end;

      if NORMAL then
         select
           accept INIT;
         or
           accept NEXT(S: out SYMBOLTYPE;
              I: out INFOTYPE;
              VALID_NEXT: out BOOLEAN;
              END_OF_ITER: out BOOLEAN) do
                 VALID_NEXT:= TRUE;
                 END_OF_ITER:= TRUE;
                 INITIALIZED:= FALSE;
           end NEXT;
         end select;
      else
         NORMAL:= TRUE;
      end if;
   end loop;
end ITERATOR_MANAGER;
```

Unfortunately there are syntactic problems with this implementation. Before dealing with these, consider the relationship between exception propagation and recursive algorithms. Notice the interaction between the boolean state variable NORMAL and the exception END_OF_ITER. NORMAL is always true unless an INIT entry is processed before an iteration (tree traversal) is completed. As soon as NORMAL is set to false the exception BREAKOUT is raised which propogates completely back through the recursive calls to ITERATE. This use of exceptions provides a clean way to "short-circuit" and "clean up" recursive algorithms and is very useful in applications such as recursive descent parser error handling.

III. Syntactic Problems with the Implementation of Section II

As stated in the previous section, the astute Ada programmer should recognize problems with the Ada implementation of the ITERATOR_MANAGER. One relatively minor problem is that a formal parameter of mode "out" must be assigned a value somewhere in the body of its procedure or accept block. In the case of the ITERATOR_MANAGER, "accept NEXT" appears a number of times, in some cases with "dummy" out parameters that have not been assigned a value. A simple solution here would simply be to assign these dummy parameters dummy values.

Unfortunately another, more complex problem exists. Consider the following excerpt from the Ada reference manual, section 9.5, paragraph 8:

> "An accept statement for an entry of a given task is only allowed within the corresponding task body; excluding within the body of any program unit that is, itself, inner to the task body; ..."

It is illegal to have an accept statement within the embedded recursive procedure ITERATE! This seems like a major problem until one realizes that the rendezvous between two tasks has an asymmetric structure. Indeed it is illegal to have an accept statement in a procedure nested within a task, but it is certainly legal to have an entry call inside the same nested procedure! We now consider this slightly altered implementation of the iterator, which leads us to a correct, if slightly sloppier solution.

In considering the use of entry rather than accept statements we realize that there is no corresponding nondeterministic choice operator for the entry call as there is for the accept statement (i.e., the selective accept). We are faced with the problem of how to choose between an INIT request and a NEXT request while recursively traversing through the tree. We solve this as follows: the ITERATOR_MANAGER (and its embedded recursive ITERATE procedure) will, at each step in the tree traversal, first find the next element and then call a middleman task to ask whether or not the element is wanted. If the element is wanted, the middleman is called again with the information. In either case, the ITERATOR_MANAGER is then free to find the next element or raise an exception to back out of the recursion. The middleman meanwhile presents to the user the abstraction described by the finite state machine in figure 4. A Buhr diagram of components is shown in figure 5.

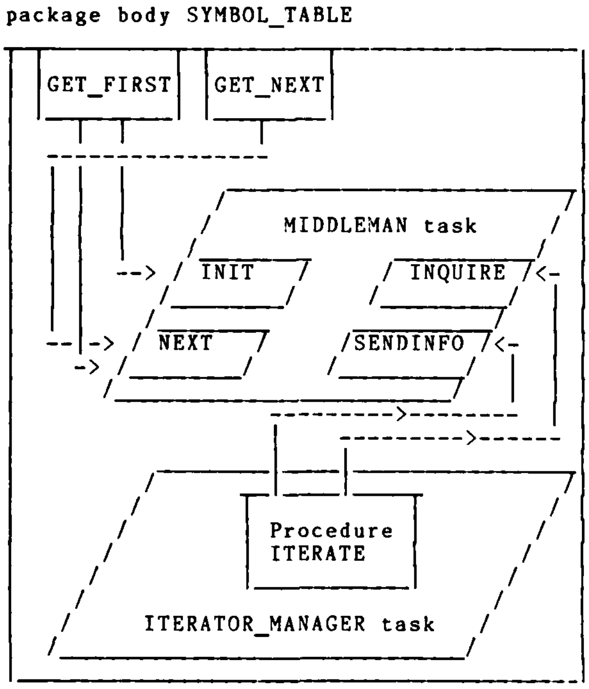


figure 5

The following is a specification of the middleman task. The finite state machine of figure 4 is now embodied in the INIT and NEXT entries of this task. Note that a new type is introduced, COMMAND, that is used to communicate the user request (INIT or NEXT) through the INQUIRE entry.

```
type COMMAND is (INIT_COM,NEXT_COM);

task MIDDLEMAN is

   -- Interface to GET_FIRST and
   --              GET_NEXT
   entry INIT;
   entry NEXT(S: out SYMBOLTYPE;
              I: out INFOTYPE;
              VALID_NEXT: out BOOLEAN;
              END_OF_ITER: out BOOLEAN);

   -- Interface to ITERATOR_MANAGER
   entry INQUIRE(C: out COMMAND);
   entry SEND_NEXT
         (SEND_S: in SYMBOLTYPE;
          SEND_I: in INFOTYPE;
          SEND_VALID_NEXT: in BOOLEAN;
          SEND_END_OF_ITER: in BOOLEAN);
end MIDDLEMAN;
```

The GET_FIRST and GET_NEXT user interface procedures are identical to those in the previous implementation except that they call MIDDLEMAN.INIT and MIDDLEMAN.NEXT instead of ITERATOR_GENERATOR.INIT and ITERATOR.NEXT.

The task body of MIDDLEMAN and the new task specification and body for the ITERATOR_MANAGER are presented next. Note that since the ITERATOR_MANAGER is now making entry calls rather than accepting them, its specification contains no entry specifications.

```
task body MIDDLEMAN is
begin
 loop
   select
     accept INIT;
       accept INQUIRE(C: out COMMAND) do
           C:= INIT_COM;
       end INQUIRE;
     or
       accept NEXT(S: out SYMBOLTYPE;
             I: out INFOTYPE;
             VALID_NEXT: out BOOLEAN;
             END_OF_ITER: out BOOLEAN) do
         accept INQUIRE(C: out COMMAND) do
           C:= NEXT_COM;
         end INQUIRE;
         accept SEND_NEXT
          (SEND_S: in SYMBOLTYPE;
          SEND_I: in INFOTYPE;
          SEND_VALID_NEXT: in BOOLEAN;
          SEND_END_OF_ITER: in BOOLEAN) do
            S:= SEND_S;
            I:= SEND_I;
            VALID_NEXT:= SEND_VALID_NEXT;
            END_OF_ITER:=SEND_END_OF_ITER;
         end SEND_NEXT;
       end NEXT;
     end select;
   end loop;
end MIDDLEMAN;
```

```
task ITERATOR_MANAGER is
end ITERATOR_MANAGER;

task body ITERATOR_MANAGER is
   NORMAL: BOOLEAN:= TRUE;
   INITIALIZED: BOOLEAN:= FALSE;
   BREAKOUT: exception;
   C: COMMAND;

   procedure ITERATE(N: in NODE) is
   begin
      if N /= null then
         ITERATE(N.LEFT);
         MIDDLEMAN.INQUIRE(C);
         if C = INIT_COM then
            NORMAL:= FALSE;
            raise BREAKOUT;
         else -- C = NEXT_COM
            MIDDLEMAN.SEND_NEXT(N.SYMBOL,
                                N.INFO,
                                TRUE,
                                FALSE);
         end if;
         ITERATE(N.RIGHT);
      end if;
   end ITERATE;

begin -- BODY OF ITERATOR_MANAGER
   loop
      while not INITIALIZED loop
         MIDDLEMAN.INQUIRE(C);
         if C = INIT_COM then
            INITIALIZED:= TRUE;
         else -- C = NEXT_COM
            MIDDLEMAN.SEND_NEXT("",
                                "",
                                FALSE,
                                TRUE);
         end if;
      end loop;

      begin
         ITERATE(ROOT);
      exception
         when BREAKOUT => null;
      end;

      if NORMAL then
         MIDDLEMAN.INQUIRE(C);
         if C = NEXT_COM then
            MIDDLEMAN.SEND_NEXT("",
                                "",
                                TRUE;
                                TRUE);
            INITIALIZED:= FALSE;
         end if;
      else
         NORMAL:= TRUE;
      end if;
   end loop;
end ITERATOR_MANAGER;
```

IV. The Quadcode Iterator

Now that we have explored the use of tasking to implement a tree structured symbol table iterator, consider the problem at hand: the organization of objects in a plane in such a way that an object can be efficiently retrieved by specifying a single coordinate point "within" the object. One way to do this is by recursively splitting the (bounded) plane into a collection of rectangles, assigning each rectangle a "quadcode", and representing an object as a subset of these quadcodes. We can then specify a single coordinate (usually through a mouse-pick), generate its quadcode, and check if this quadcode is in an object quadcode subset.

More specifically, quadcodes are assigned in the following manner. A rectangle is split into quadrants with each quadrant given a code from 0 to 3 clockwise from the upper left. Each quadrant in turn is split into quadrants, with digits 0 to 3 appended again clockwise from the upper left. The splitting process can be arbitrarily deep, with the number of digits in each quadcode corresponding to the number of times the rectangle is split. Figure 6 shows a rectangle split twice, with quadcodes of length 2.

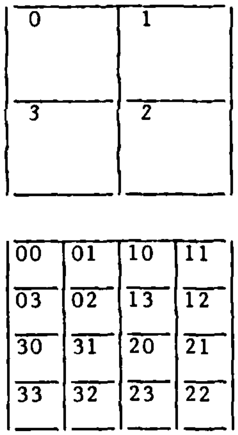


figure 6

Now, consider the following two rectang

1. A world with lower left x,y coordinates 0,0 and upper right x,y coordinates 40,40, and

2. A source rectangle in the world with lower left x,y coordinates 7.5,27.5 and upper right x,y coordinates 17.5,37.5.

A diagram of these two rectangles appears in figure 7.

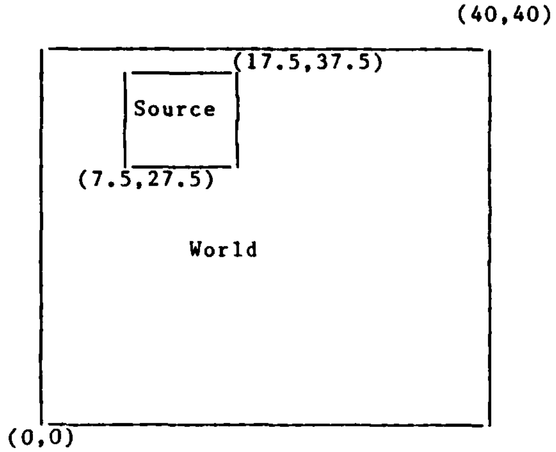


figure 7

If we recursively split the world into quadrants and assign the corresponding quadcodes, we derive the diagram in figure 8.

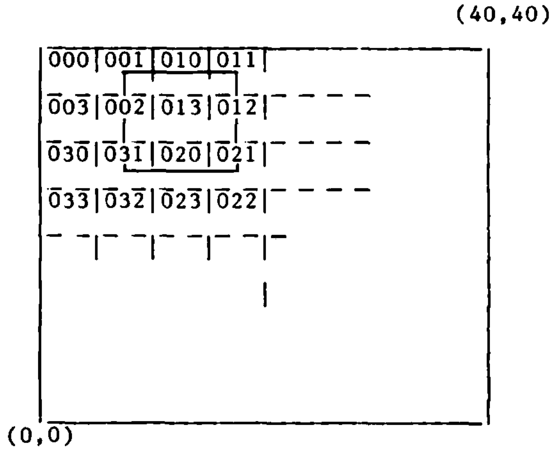


figure 8

Notice that the source rectangle overlaps a number of quadrants. Specifically, it's overlapped quadrant set, represented by a set of quadcodes, is:

{ 001, 002, 010, 011, 012, 013, 020, 021, 031 }

We use an iterator to generate a subset of quadcodes representing an object such as the one in the previous example (i.e., source), storing these quadcodes in a B-tree index for subsequent look-up. Since quadcodes can be determined by a recursive splitting algorithm, we use this algorithm as the basis for our iterator, just as we used the recursive in-order traversal algorithm for the symbol table iterator.

Let us first assume that the following RECTANGLES package is available:

```
package RECTANGLES is
   type RECTANGLE is private;
        .
        .
   type QUADRANT is (UPPER_LEFT,
                     UPPER_RIGHT,
                     LOWER_RIGHT,
                     LOWER_LEFT);
   function QUADRANT_OF(R: RECTANGLE;
                        C: QUADRANT)
                         return RECTANGLE;
   function IS_IN(R1,R2: RECTANGLE)
                         return BOOLEAN;
        .
private
   {type RECTANGLES is implementation
    dependent}
end RECTANGLES;
```

We now present the specification of the QUAD_CODE iterator package. Notice that since the initialization step of the iterator must be provided with three parameters, i.e., the source, world, and depth of resulting quadcodes, we will not provide a GET_FIRST operator. As long as INITIALIZE_ITERATOR has been called, the first call to NEXT_QC will perform the GET_FIRST operation.

```
with RECTANGLES;
use RECTANGLES;
package QUAD_CODES is

   procedure INITIALIZE_ITERATOR
                   (MAX: in POSITIVE;
                    SOURCE: in RECTANGLE;
                    WORLD: in RECTANGLE);

   DEPTH_OVERFLOW: exception;

   function NEXT_QC return NATURAL;

   UNINITIALIZED_ITERATOR,
   END_OF_ITERATION: exception;

end QUAD_CODES;
```

A code segment to generate all 4 digit quadcodes of a rectangle SOURCE in a rectangle WORLD might be implemented as follows:

```
begin
   INITIALIZE_ITERATOR(4,SOURCE,WORLD);
   loop
      QUAD_CODE:= NEXT_QC;

      -- process this QUAD_CODE

   end loop;
```

```
exception
   when END_OF_ITERATION =>
      -- end of QUAD_CODE processing
end;
```

Now for the implementation of the QUAD_CODES package. The interface operations INITIALIZE_ITERATOR (procedure) and NEXT_QC (function) are similar in structure to the GET_FIRST and GET_NEXT operations in the previous section and are presented below.

```
procedure INITIALIZE_ITERATOR
                  (MAX: in POSITIVE;
                   SOURCE: in RECTANGLE;
                   WORLD: in RECTANGLE) is
begin
   if MAX > 9 then
      raise DEPTH_OVERFLOW;
   end if;
   MIDDLEMAN.INIT(MAX,SOURCE,WORLD);
end INITIALIZE_ITERATOR;

function NEXT_QC return NATURAL is
   QC: NATURAL;
   VALID: BOOLEAN;
   EOF: BOOLEAN;
begin
   MIDDLEMAN.NEXT(QC,VALID,EOF);
   if not VALID then
      raise UNINITIALIZED_ITERATOR;
   end if;
   if EOF then
      raise END_OF_ITERATION;
   end if;
   return QC;
end NEXT_QC;
```

The remainder of the implementation, the MIDDLEMAN and ITERATOR_MANAGER task specifications and bodies, are presented next.

```
type COMMAND is (INIT_COM,NEXT_COM);

task MIDDLEMAN is

   -- Interface to GET_FIRST and
   --                GET_NEXT
   entry INIT(MAX: in POSITIVE;
              SOURCE: in RECTANGLE;
              WORLD: in RECTANGLE);
   entry NEXT(QC: out NATURAL;
              VALID_NEXT: out BOOLEAN;
              END_OF_ITER: out BOOLEAN);


   -- Interface to ITERATOR_MANAGER
   entry INQUIRE(C: out COMMAND);
   entry GET_INIT
         (SEND_MAX: out POSITIVE;
          SEND_SOURCE: out RECTANGLE;
          SEND_WORLD: out RECTANGLE);
   entry SEND_NEXT
         (SEND_QC: in NATURAL;
          SEND_VALID_NEXT: in BOOLEAN;
          SEND_END_OF_ITER: in BOOLEAN);
end MIDDLEMAN;


task body MIDDLEMAN is
begin
 loop
   select
     accept INIT(MAX: in POSITIVE;
                 SOURCE: in RECTANGLE;
                 WORLD: in RECTANGLE) do
       accept INQUIRE(C: out COMMAND) do
           C:= INIT_COM;
       end INQUIRE;
       accept GET_INIT
          (SEND_MAX: out POSITIVE;
           SEND_SOURCE: out RECTANGLE;
           SEND_WORLD: out RECTANGLE) do
              SEND_MAX:= MAX;
              SEND_SOURCE:= SOURCE;
              SEND_WORLD:= WORLD;
       end GET_INIT;
     end INIT;
   or
     accept NEXT
         (QC: out NATURAL;
          VALID_NEXT: out BOOLEAN;
          END_OF_ITER: out BOOLEAN) do
       accept INQUIRE(C: out COMMAND) do
         C:= NEXT_COM;
       end INQUIRE;
       accept SEND_NEXT
        (SEND_QC in NATURAL;
        SEND_VALID_NEXT: in BOOLEAN;
        SEND_END_OF_ITER: in BOOLEAN) do
            QC:= SEND_QC;
            VALID_NEXT:= SEND_VALID_NEXT;
            END_OF_ITER:=SEND_END_OF_ITER;
       end SEND_NEXT;
     end NEXT;
   end select;
 end loop;
end MIDDLEMAN;


task ITERATOR_MANAGER is
end ITERATOR_MANAGER;


task body ITERATOR_MANAGER is
   NORMAL: BOOLEAN:= TRUE;
   INITIALIZED: BOOLEAN:= FALSE;
   BREAKOUT: exception;
   C: COMMAND;

   MAX_DEPTH: POSITIVE;
   SOURCE,WORLD: RECTANGLE;


procedure ITERATE(QUAD: RECTANGLE;
                  QUAD_DEPTH: POSITIVE;
                  CUR_QC: NATURAL) is
begin
   if QUAD_DEPTH > MAX_DEPTH then
      MIDDLEMAN.INQUIRE(C);
      if C = INIT_COM then
         MIDDLEMAN.GET_INIT(MAX_DEPTH,
                            SOURCE,
                          WORLD);
         NORMAL:= FALSE;
         raise BREAKOUT;
      else
         MIDDLEMAN.SEND_NEXT(CUR_QC,
                             TRUE,
                             FALSE);
      end if;
   else
      for Q in QUADRANT loop
        LOWER_QUAD:= QUADRANT_OF(QUAD,Q);
        if IS_IN(SOURCE,LOWER_QUAD) then
           ITERATE(LOWER_QUAD,
                   QUAD_DEPTH+1,
                   CUR_QC*10 +
                   QUADRANT'POS(Q));
        end if;
      end loop;
   end if;
end ITERATE;

begin -- BODY OF ITERATOR_MANAGER
   loop
      while not INITIALIZED loop
         MIDDLEMAN.INQUIRE(C);
         if C = INIT_COM then
            INITIALIZED:= TRUE;
            MIDDLEMAN.GET_INIT(MAX_DEPTH,
                               SOURCE,
                               WORLD);
         else -- C = NEXT_COM
            MIDDLEMAN.SEND_NEXT(0,
                                FALSE,
                                TRUE);
         end if;
      end loop;

      begin
         ITERATE(WORLD,1,0);
      exception
         when BREAKOUT => null;
      end;

      if NORMAL then
         MIDDLEMAN.INQUIRE(C);
         if C = NEXT_COM then
            MIDDLEMAN.SEND_NEXT(0,
                                TRUE;
                                TRUE);
            INITIALIZED:= FALSE;
         end if;
      else
         NORMAL:= TRUE;
      end if;
   end loop;
end ITERATOR_MANAGER;
```

V. Further Notes on Iterators

Notice that in the tree structured SYMBOL_TABLE iterator the order in which the symbols were retrieved was alphabetical. It is important to note here that this order is implementation dependent and not necessarily a property of the abstraction. In fact, no information is given in the Ada package specification to clarify this point. If an alphabetical ordering was indeed desired, the following addition to the specification would be in order.

```
package SYMBOL_TABLE is
    .
    .

-- Iterator operations and exceptions:
--    this iterator will return
--    symbols of the table in
--    alphabetical order

procedure GET_FIRST(SYM: out SYMBOLTYPE;
                    INFO: out INFOTYPE);
procedure GET_NEXT(SYM: out SYMBOLTYPE;
                   INFO: out INFOTYPE);
  END_OF_ITERATION,
  UNINITIALIZED_ITERATOR: exception;

end SYMBOL_TABLE;
```

Consider now the way that the implementation described in section two utilizes the tree structure. Implementation details of the tree structure (a dynamically allocated structure of nodes) are directly visible within the package body of SYMBOL_TABLE. Suppose on the other hand that a tree abstraction was first implemented and then used to implement the symbol table body. A number of choices then exist for implementing the symbol table iterator.

1. The symbol table iterator could be implemented on top of the tree abstraction, or

2. An iterator could be provided by the tree abstraction itself, with the symbol table iterator simply a renaming of the tree iterator operations.

In the latter case, a number of iterators could be provided by the tree abstraction: an inorder iterator, a pre-order iterator, a post-order iterator, an iterator that returns nodes in arbitrary order, etc. One reason why an iterator that returns nodes in arbitrary order might be useful is that the user might be concerned solely with the collection of nodes, and not with the parent/child relationships between the nodes. The implementor of the tree would then be free to take advantage of this arbitrary ordering (and would be expected to do so) to return nodes in a more efficient manner than could be done by traversing the tree in a structured way.

VI. Summary and Conclusions

We have presented what we feel to be an interesting implementation of an iterator for data structures that can be most neatly expressed using recursion. The questions of the efficiency of both the embedded recursive ITERATE procedure and the multi-tasking implementation (requiring two additional tasks for each iterator) were not of primary concern to us, since other bottlenecks in our system overshadowed any slowdowns due to the iterator. If task and recursion overhead were indeed a problem, two alternate ways to implement the iterators presented would be to either provide a threaded tree or explicit run-time stack, and write iterative procedures for tree traversal. The state of the iteration could then be saved between each call to the iterator.

This work has grown, as has our work on type management schemes in last year's conference, out of the work of our Computer Science Department Advanced Projects Lab. We are developing SEE, a learning environment toolset (IEEE Conf. on Ada Applications and Environments, April, 1986), of which SeeGraph is a part. A word of acknowledgment is in order for Ms. Elizabeth Johnson, who spent long hours on SeeGraph and who has been kind enough to test the code presented in this paper.

References

1. Atkinson, Russell R., Liskov, Barbara H., and Schleifler, Robert W., "Aspects of implementing CLU", Proceedings of the ACM 1978 Annual Conference.

2. Booch, Grady, Software Components with Ada - Structures, Tools, and Subsystems, Benjamin Cummings, 1987.

3. Buhr, R.J.A., System Design with Ada, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

4. Guttag, John, and Liskov, Barbara H., Abstraction and Specification in Program Development, The MIT Press, 1986.

5. Latour, Larry, "A Programming Environment for Learning - SEE: A Student's Educational Environment", Proceed.ngs of the IEEE Conference on Ada Applications and Environments, Miami Beach, FL., 1986.

6. Latour, Larry, and Johnson, Elizabeth, "SeeGraph: A Description of a Graphic Based Knowledge Representation System", Internal Report, University of Maine Computer Science Dept., Orono, ME., 1987.

7. Li, Shu-Xiang, and Loew, Murray H., "Adjacency Detection Using Quadcodes", Communications of the ACM, Vol. 30, No. 7, July, 1987.

8. Li, Shu-Xiang, and Loew, Murray H., "The Quadcode and Its Arithmetic", Communications of the ACM, Vol. 30, No. 7, July, 1987.

9. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815 A, United States Department of Defense, 1983.

10. Samet, Hanan., "The Quadtree and Related Hierarchical Data Structures", ACM Computing Surveys, Vol. 16, No. 2, June, 1984.

11. Shaw, Mary (editor), Alphard: Form and Content, Springer-Verlag New York, Inc., 1981.

12. Shaw, Mary, Wulf, Wm. A., and London, Ralph L., "Abstraction and Verification in Alphard: Iteration and Generators", Carnegie-Mellon University and USC Information Sciences Institute Technical Reports, 1976.

Larry Latour received the B.B.A Degree in Statistics from Baruch College/CUNY in 1973, the M.S. Degree in Operations Research from Polytechnic Institute of New York in 1978, and the Ph.D. Degree in Computer Science from Stevens Institute of Technology in 1985. His research interests include database transaction systems, learning and reusability environments, and knowledge representation tools. He is currently an Asst. Professor of Computer Science at the University of Maine, Orono, ME.

Programming with Streams in Ada

Reginald N. Meeson, Jr.

Institute for Defense Analyses
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-3541

Abstract

Streams are abstract datatypes that generalize the concept of sequential data structures such as arrays, linked lists, sequential files, and communications between tasks. Streams significantly enhance the modularity and reusability of Ada program components. Programs constructed using streams can be easily reconfigured for new applications and for testing by substituting stream sources, by exchanging processing modules, or by rerouting stream flows. Streams support "programming in the large" by hiding the details of procedural control flow and by highlighting the data that flows between system components. This paper briefly describes a collection of generic packages that have been built to support stream declarations and processing. The complete set of these packages includes primitive operations, utility functions, and higher-level generic operators that combine to form a powerful program development technique.

Introduction

Streams are sequential data structures of arbitrary length. They arise naturally from many sources. For example, sequential input and output are common forms of streams. Many iterative processes can be thought of as operating on streams of data. Programming with streams has been described for other environments [1-4]. This is the first treatment I am aware of for Ada.

Streams are key components in popular design techniques such as data flow design [5]. Data flows that carry multiple instances of data objects are typically interpreted as streams. Streams are more like files than they are like arrays or linked lists, since they can be much larger than ordinary program data objects. Unlike files, though, which are external data objects, streams are internal objects that are used to transfer data between program components.

A model for how streams can be used in developing large software systems is provided by the Unix pipe [6]. Pipes in Unix allow programs to communicate with each other through standard sequential file interfaces. Simple programs can easily be connected together to perform more complex and useful functions using pipes. The separate component programs are completely independent. They do not rely on any knowledge about the existence or operation of other programs with which they might be combined. Their interaction is set up and controlled entirely by the pipe mechanism. Streams in Ada provide similar advantages for constructing programs from reusable components. In addition, Ada provides streams of complex objects (pipes are limited to byte streams) and complete type checking.

Since streams, typically, will be too large to be produced in their entirety and held in program memory, stream elements must be produced incrementally. This allows stream elements to be generated on demand as they are needed, which is a form of "lazy evaluation" [7]. Lazy evaluation can be emulated in Ada by providing procedures to advance streams and produce their consecutive elements. In the approach I have taken, therefore, streams are not really data structures, they are procedures.

Ideally, I would like to be able to create streams and pass them between program units like data objects. This would allow me to directly implement high-level designs. The approach I have taken provides most of the desirable features of streams, even though Ada does not support creating and passing procedures as data. Hence, I will continue to describe streams as objects that can be passed between and operated on by program units, and I will use the terms "stream objects" and "stream types" as metaphors.

When I speak of creating a new stream object, I mean I am defining its generating procedure. Similarly, when I refer to stream types, I mean the class of procedures that generate streams with elements of a particular type. Ada's generics support these metaphors. Below I give examples of how new streams can be created by generic package instantiations and how complete programs can be constructed from component parts.

Primitive Stream Operations

The fundamental primitive operation on a stream object is its generating procedure. The key information that this procedure must produce is an end-of-stream indicator and the next stream element (if there is one). There is also an optional input parameter that allows a stream-consuming process to signal the generator when no more stream elements will be requested. The canonical stream generating procedure, therefore, looks like this:

```
procedure ADVANCE (EOS: out BOOLEAN;
                   NEXT: out STREAM_ELEMENT;
                   MORE: in BOOLEAN := TRUE );
```

By convention, stream-generating procedures are encapsulated in packages. This provides a mechanism for naming streams (the package name) and provides storage for essential stream state data that must be retained between procedure calls.

Utility Operations

Beyond primitive stream-generating procedures, there are a number of convenient utility operations on streams. These include: concatenating streams, merging streams, and converting sequential files into streams and back again. Stream concatenation, for example, can be implemented by defining a new generating procedure that operates by:

(1) Advancing the first input stream and reproducing its results until it ends, and then

(2) Switching to the second input stream and reproducing its results.

The process that consumes the concatenated stream is entirely independent of this activity and will not recognize when the transition occurs. I have generalized this process so that two streams can be concatenated by instantiating a single generic stream package. The specification for this package is:

```
generic
   type STREAM_ELEMENT is private;
   with procedure FIRST_ADVANCE (
                      EOS: out BOOLEAN;
                      NEXT: out STREAM_ELEMENT;
                      MORE: in BOOLEAN := TRUE );
   with procedure SECOND_ADVANCE (
                      EOS: out BOOLEAN;
                      NEXT: out STREAM_ELEMENT;
                      MORE: in BOOLEAN := TRUE );
package CONCATENATION is
   procedure ADVANCE ( EOS: out BOOLEAN;
                  NEXT: out STREAM_ELEMENT;
                  MORE: in BOOLEAN := TRUE );
end CONCATENATION;
```

Input and output of streams as sequential files are natural conversions between internal and external forms of the same objects. The following declaration specifies a generic sequential file input handler that converts a file into a stream. Each time one of these streams is advanced, another element is read from the file and returned.

```
generic
   type STREAM_ELEMENT is private;
   type SEQUENTIAL_FILE is limited private;
   INPUT: in out SEQUENTIAL_FILE;
package INPUT_STREAM is
   procedure ADVANCE ( EOS: out BOOLEAN;
                  NEXT: out STREAM_ELEMENT;
                  MORE: in BOOLEAN := TRUE );
end INPUT_STREAM;
```

The complement of this package is an output driver that consumes a stream and produces a sequential output file. The specification for a generic procedure that converts a stream into a sequential file is:

```
generic
   type STREAM_ELEMENT is private;
   type SEQUENTIAL_FILE is limited private;
procedure OUTPUT_DRIVER (
             OUTPUT: in out SEQUENTIAL_FILE );
```

Only one call to an output driver is required to produce the entire output file. This is consistent with our objective of creating high-level operations that operate on entire streams.

A Sample Stream Program

The stream operations described so far can be used to create a very simple example that demonstrates the concept of constructing stream programs. Below is an outline of a program that concatenates two sequential files.

```
procedure SAMPLE_STREAM_PROGRAM is
   type SAMPLE_ELEMENT is ... ;
   type SAMPLE_SEQ_FILE is ... ;
   FILE_A, FILE_B, FILE_C: SAMPLE_SEQ_FILE;
   package STREAM_A is new
      INPUT_STREAM( SAMPLE_ELEMENT,
         SAMPLE_SEQ_FILE, FILE_A );
   package STREAM_B is new
      INPUT_STREAM( SAMPLE_ELEMENT,
         SAMPLE_SEQ_FILE, FILE_B );
   package STREAM_C is new
      CONCATENATION( SAMPLE_ELEMENT,
         STREAM_A.ADVANCE,
         STREAM_B.ADVANCE );
   procedure PRODUCE is new
      OUTPUT_DRIVER( SAMPLE_ELEMENT,
         SAMPLE_SEQ_FILE );
begin
   OPEN( FILE_A, ... );
   OPEN( FILE_B, ... );
   CREATE( FILE_C, ... );
   PRODUCE( FILE_C );
end SAMPLE_STREAM_PROGRAM;
```

As you can see, the program consists almost entirely of declarations and generic instantiations. The only variables declared are for the three files. All of the other necessary variables are hidden within the generic units. The first two generic package instantiations create the input streams and the third creates the concatenated output stream. The generic procedure instantiation creates the procedure that produces the output file.

Higher-Level Stream Operations

In addition to utility functions, there are a number of useful, higher-level operations that perform more substantive stream transformations. The first example, called "mapping," applies a function to each element of an input stream and produces a new stream containing the function's results. There are numerous applications for this operation. One very simple example is to convert an input stream of EBCDIC characters into an ASCII character stream. The distinguishing characteristics of mappings are that they produce one-to-one images of their input streams, and that each output element depends only on the value of the corresponding input element. The specification for the stream mapping package is:

```
generic
   type INPUT_ELEMENT is private;
   type OUTPUT_ELEMENT is private;
   with function F ( X: in INPUT_ELEMENT )
      return OUTPUT_ELEMENT;
   with procedure INPUT_ADVANCE (
                        EOS: out BOOLEAN;
                        NEXT: out INPUT_ELEMENT;
                        MORE: in BOOLEAN := TRUE );
package MAPPING is
   procedure ADVANCE ( EOS: out BOOLEAN;
                       NEXT: out OUTPUT_ELEMENT;
                       MORE: in BOOLEAN := TRUE );
end MAPPING;
```

Another common high-level stream operation is called "filtering." Filters require a Boolean test function that can be applied to each element of an input stream. If an element passes the test, it is returned as the next output element. If it fails, it is thrown away and another element is drawn from the input stream and tested. The output stream, therefore, contains only those input elements that pass the test. The specification for the stream filtering package is:

```
generic
   type STREAM_ELEMENT is private;
   with function TEST ( X: in STREAM_ELEMENT )
      return BOOLEAN;
   with procedure INPUT_ADVANCE (
                        EOS: out BOOLEAN;
                        NEXT: out STREAM_ELEMENT;
                        MORE: in BOOLEAN := TRUE );
package FILTERING is
   procedure ADVANCE ( EOS: out BOOLEAN;
                       NEXT: out STREAM_ELEMENT;
                       MORE: in BOOLEAN := TRUE );
end FILTERING;
```

In addition to mapping and filtering operations, there are several common types of finite-state machine processes that operate on streams. One example, which is called a "transducer," emits the stream of states that it passes through as it consumes its input stream. The first output element produced is the machine's initial state. Then, as each successive output element is requested, a new input element is fetched and the current state is updated to form a new state. Each new state is returned as an output stream element. Examples of transducers include Kalman filters and other signal processing applications. The specification for the stream transducer package is:

```
generic
   type INPUT_ELEMENT is private;
   type MACHINE_STATE is private;
   INITIAL_STATE: MACHINE_STATE;
   with procedure UPDATE (
                        INPUT: in INPUT_ELEMENT;
                        CURRENT_STATE: in out
                           MACHINE_STATE );
   with procedure INPUT_ADVANCE (
                        EOS: out BOOLEAN;
                        NEXT: out INPUT_ELEMENT;
                        MORE: in BOOLEAN := TRUE );
package TRANSDUCER is
   procedure ADVANCE ( EOS: out BOOLEAN;
                       NEXT: out MACHINE_STATE;
                       MORE: in BOOLEAN := TRUE );
end TRANSDUCER;
```

There are many variations of finite-state machine processes that consume inputs and produce outputs at different rates. A simple example is a text formatter that turns a stream of strings and formatting commands into a stream of characters for printing. Sorting is another example. For sorting, the entire input stream must be consumed before the first output element can be produced.

A variation of a finite-state machine process that returns only its final state value is called a "reduction." A simple example of a reduction is a process that forms the sum of a stream of numbers. The specification for the generic reduction procedure is:

```
generic
   type STREAM_ELEMENT is private;
   type MACHINE_STATE is private;
   INITIAL_STATE: MACHINE_STATE;
   with procedure UPDATE (
                        INPUT: in STREAM_ELEMENT;
                        CURRENT_STATE: in out
                           MACHINE_STATE );
   with procedure INPUT_ADVANCE (
                        EOS: out BOOLEAN;
                        NEXT: out INPUT_ELEMENT;
                        MORE: in BOOLEAN := TRUE );
procedure REDUCTION (
              FINAL_STATE: out MACHINE_STATE );
```

Another variation of a finite-state machine process is one that produces an infinite sequence of self-generated states. An example of such a process is a random number generator that produces an infinite stream of (pseudo)random numbers. The initial state is the "seed" for the numbers generated and the update procedure determines the distribution of random values produced. The specification for the stream generator package is:

```
generic
   type MACHINE_STATE is private;
   INITIAL_STATE: MACHINE_STATE;
   with procedure UPDATE (
                        CURRENT_STATE: in out
                           MACHINE_STATE );
package GENERATOR is
   procedure ADVANCE ( EOS: out BOOLEAN;
                       NEXT: out MACHINE_STATE;
                       MORE: in BOOLEAN := TRUE );
end GENERATOR;
```

Shared Streams and Side Effects

Common object-oriented data flow design techniques assume that streams can be freely split or shared between consuming processes. This is highly desirable, but there are several interpretations of what splitting a stream means. If multiple processes invoke one of the stream-generating procedures I have described, each will receive only a subset of the stream's elements, not a complete copy of the stream. A stream of numbers shared between a counting process and a summing process that combine their results to compute an average, for example, would not produce correct results.

Stream-sharing processes must be able to synchronize their "advance" operations to avoid the destructive side effects caused by generating the next input stream element. Our solution to this is to create a task with two entries (for a two-way split) that look exactly like stream-generating procedures. The task's job is to control the progress of the two consuming processes and to distribute copies of the input stream elements appropriately. The task may also buffer a number of input elements. If one consuming process decides that it will not need any more stream elements, it is required to notify the task (by setting its "more" parameter to "false") so that the other process can be allowed to continue without further interference. The specification for this stream-sharing task is:

```
generic
   type STREAM_ELEMENT is private;
   BUFFER_SIZE: INTEGER;
   with procedure INPUT_ADVANCE (
                     EOS: out BOOLEAN;
                     NEXT: out STREAM_ELEMENT;
                     MORE: in BOOLEAN := TRUE );
package SHARE is
   task ADVANCE is
      entry COPY_1 ( EOS: out BOOLEAN;
               NEXT: out STREAM_ELEMENT;
               MORE: in BOOLEAN := TRUE );
      entry COPY_2 ( EOS: out BOOLEAN;
               NEXT: out STREAM_ELEMENT;
               MORE: in BOOLEAN := TRUE );
   end ADVANCE;
end SHARE;
```

Conclusion

I have introduced the concept of stream datatypes and briefly described a collection of generic operations on streams. Using these operations as program construction tools, program components can be created and tested quickly, and easily integrated into complete programs. Stream programs can be easily reconfigured for new applications and for testing by substituting stream sources, by exchanging processing modules, or by rerouting stream flows.

The generic stream packages described above have been implemented on a VAX 8600 using the DEC Ada compiler, and have been ported to a Sun 3/50 using the Verdix Ada compiler. The tasking overhead for shared streams is tolerable for rapid prototyping applications and for exercising executable specifications. I expect performance can be improved to enable the use of streams in general (i.e., non-time-critical) applications.

References

1. Burge, W.H., Recursive Programming Techniques, Addison-Wesley, 1975.
2. Goldberg, A. and R. Paige, "Stream Processing," 1984 ACM Symp. on Lisp and Functional Programming, August 1984, pp 53-62.
3. Goldberg, A. and D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.
4. Ida, T. and J. Tanaka, "Functional Programming with Streams," Proc. IFIP '83, North-Holland, 1983.
5. DeMarco, T., Structured Analysis and System Specification, Prentice-Hall, 1979.
6. Ritchie, D. and K. Thompson, "The UNIX Time-Sharing System," Comm. ACM, 17, 7, July 1974, pp. 365-375.
7. Henderson, P. and J. H. Morris, "A Lazy Evaluator," Proc. 3rd ACM Symp. on Principles of Programming Languages, 1976, pp. 95-103.

LESSONS LEARNED IN ADA* TRAINING

ELIZABETH A. KENNEDY

ROCKWELL INTERNATIONAL, STSD

ABSTRACT

In 1985, Rockwell Divisions in Southern California initiated an Ada training program. While the development of the Ada training program was a cooperative effort among the divisions each division conducted its own Ada training program. A total of 393 Rockwell engineers including 125 from the Space Transportation Systems Division (STSD) have completed this program. The program was rated as excellent by the California State Employment Development Office. The STSD program is now being revised to incorporate the experience gained in teaching Ada over the past two years. In the paper that follows, the recommendations for the STSD program are presented to assist persons involved in Ada and Software Engineering training programs.

THE FIRST GENERATION TRAINING PROGRAM

The Rockwell Ada instructional program was initiated in 1985 by establishing a "train the trainers" activity. A limited number (20) of employees attended the class conducted by a professional Ada training company. The contract provided not only for the initial instruction but also for Rockwell's right to subsequently use the presented material. The 80 hour course had a 10 to 1 student to teacher ratio. The students, knowing that they were to be teachers, were highly motivated. In addition, the well prepared vendor and the the commitment of Rockwell in putting twenty of its employees on temporary full time training assignment contributed to producing a successful "train the trainers" program. The initial activity was completed in two weeks after which the Ada instructional regimen for the engineering staff was developed.

The STSD in-house class structure was half lecture, and half lab, two hours each. The responsibility for the twenty-five lectures was distributed among several instructors allowing each lecturer time to prepare review quizzes and lab assignments in addition to regular work assignments. An informal survey of one class showed that many students liked the variety of lecturers. The materials used during the two hour lab were assignments developed by the instructor and a computer based instruction (CBI) product. The CBI product allowed the student to work independently. The assignments insured that the students learned how to use the Ada development tools. During lab, the 30 students worked in pairs, each pair working at its own pace with two to three lab assistants available to answer questions. Fifteen terminals in one room were dedicated to the students. The more advanced students completed all CBI assignments, deriving several different solutions to each problem as they experimented with the language. The beginning students were able to spend more time on each assignment without necessarily completing all assignments. They used the time to ask questions of the lab instructors and review the subjects covered in lecture. Students who failed to attend lecture or lab were expected to make up the time by watching video tapes, reading Ada text books or by working lab assignments. Exams were given at midterm and at the end of class. The examinations provided the students a quantitative measure of their progress as well as notifying the training staff of instructional difficulties and problems students were experiencing. Letter grades were assigned to these exams as a motivation factor. The students realized that those receiving the highest grades were most likely to receive Ada programming assignments.

Support from several levels of management was a major factor in the success of the Ada training program. The Training Department was responsible for contracts with the State of California and vendors. It also obtained funding for all materials and coordinated with other Rockwell divisions. Each Systems Engineering Director provided a representative to a training committee which provides a focal point in formulating requirements for technical enhancement training needs. This committee served as a central communication point for establishing student rosters, performing student administrative functions, supporting teachers in the resolution of problems, class scheduling, obtaining resources and coordinating the graduation ceremonies. The Assistant Chief Engineer of Systems Engineering attended opening day of each class to inform students and their management of the importance of Ada to Rockwell's new business pursuits. He was also present at graduation to personally congratulate each student. Each of the directors, managers and supervision of the students also attended opening day and graduation. Line management support kept the training project at the same priority as other projects and resulted in a low absenteeism rate for students and teachers.

Rockwell obtained California state support for the training program in the amount of 2000 dollars per student. The California State Employment Development Office uses its funds to pay for training that will help people keep their jobs. Commercial businesses have been involved in this program for some time, however aerospace had not participated prior to 1985. Rockwell was able to show the state that Ada would play an important part in bringing contracts into southern California, providing more jobs. State auditors visited the Ada class twice each term; they were shown the attendance and assignment charts, the lecture material and demonstrations of the CBI tool. They interviewed several students from each class and then discussed with the lead trainer suggestions for improving the class. State funding has contributed to the longevity of this in-house training program.

THE SECOND GENERATION TRAINING PROGRAM

As the STSD Ada training program is revised to bring it up to date with today's experience and technology, changes are being made to eliminate unsuccessful approaches.

Distributing the teaching responsibility among too many people caused communication problems. While some students enjoyed the variety of lectures, there was a great deal of effort spent in coordination between the instructors/lab assistants. Multiple lecturers resulted in loss of continuity; students' questions on a previously presented subject could not always be answered satisfactorily since the subject would have been taught by a different instructor. Each lecture stood alone so there were not enough connecting bridges between the subject matter covered which would have made Ada easier to learn. The next Ada class will have only four instructors. These instructors have a wide variety in work experience and more importantly their involvement in the Ada training class covers the full range from the first "train the trainers" to the most recent class. The teaching task being a larger percentage (25%) of each employee's total work task will provide more of a focus on the training task.

The decision to group students of all experience levels together in the first generation classes was based on the assumptions that having advanced students in class would improve the quality of instruction. The experienced students would be likely to provide creative solutions to assignments that could be shared with other students. They would also contribute by expressing alternate explanations of concepts. While many benefits accrued to the class as expected, some students had trouble throughout the class because they did not understand the basic concepts and the advanced students were not always stimulated by the presented material. The difficulties encountered by a "beginner" student were sometimes masked by the more experienced lab partner. During development of the second generation training program, the aim will be to produce a program that can be either a self study program or lecture program depending on the need of the student. Assignments will be developed which have beginning and advanced levels so that each type of student will be challenged rather than overwhelmed or bored. As the material for each lecture is completed the lecture will be video taped in a studio. Review and revision is scheduled to assure a quality video. Another CBI product is being purchased to organize and maintain all assignments and subject outlines developed by the instructors. This will standardize the format of the class material and will provide an efficient technique for electronic storage and retrieval of the class material. The tool will help consolidate teaching material and allow it to be easily retained as instructors change. The video tape and CBI tool material will provide some of the major materials needed for a self study program.

Examples and assignments used during the original classes were academic or commercial in nature. Students objected to working problems which did not directly relate to Rockwell's business base. In revising the class, emphasis is being placed on developing examples and lab assignments that reinforce the point of the examples. Class assignments that produce code used in projects outside the class are being considered. These projects will be presented to a group of project leaders to ensure that the assignments given in class correspond to tasks that project leads want their employees to be able to perform. To improve the practical skill level of the Ada class graduate the scope of the class is being reduced in scope to a specific Ada programming class. The objective of the class is "to graduate students who shall be able to write, edit, compile and debug modular Ada programs and small systems that can be merged into any larger Ada project required by management".

CONCLUSION

The problems encountered in the Ada training program are not unique to Ada. They exist in any training program for a new technology. The most essential element to a successful program is management support in providing resources. Keeping ideas that have proven successful, while incorporating new ideas, will result in the desired "new and improved" Ada training program. The Ada training program is a significant part of Rockwell's continued efforts to upgrade skills and stay current in technological advances. The Ada experience level has increased dramatically in the past three years allowing STSD to pursue major Ada contracts.

*Ada is a registered trademark of the U.S. Government, Ada Joint Program Office (AJPO)

BIOGRAPHY

Ms. Kennedy has participated in Rockwell's Space Transportation Systems Division's (STSD) Ada Training Program since its inception in 1984. She is currently responsible for establishing efficient software development methods within STSD's Software Engineering Department. Since 1977 she has been involved in the Space Shuttle Program by supporting launch operations at Kennedy Space Center and the Backup Flight System in Downey, California. Ms. Kennedy graduated from Florida Southern College in Lakeland, Fla. with a B.S. in Math in 1977.

Elizabeth A. Kennedy
Rockwell International
Space Transportation Systems Division
12214 Lakewood Blvd.,
Downey, Calif. 90241
(213) 922-5573

Lessons Learned Teaching Ada in the Context of Software Engineering

by

James E. Tomayko
The Wichita State University

Abstract

Educators across the country are struggling with difficult issues in to the teaching of Ada[tm] and its relationship to the Computer Science Curriculum. By design, the language supports software engineering principles. Therefore, it would seem that the "natural" place for teaching Ada is within the context of software engineering. This paper reports on the author's and his students' experiences in learning and using Ada in different settings, including a software engineering project course and a course centered on Ada and its use.

The Place of Ada in the Curriculum

Ada is unique in that it was designed to embody software engineering principles in a language aimed at a wide variety of applications domains. As such, it provides both an opportunity and a dilemma for educators. The opportunity is to take advantage of Ada's intent by introducing software engineering at an earlier stage of the curriculum. Obviously, Ada has all the advantages of Pascal or Modula II as a beginning language: structure, modularity, and few strange syntactic conventions. Anyone who has seen Booch's *Software Components with Ada* [Booch87] realizes that Ada is a suitable vehicle for teaching the essential data structures and algorithms found in the ACM Curriculum CS2 course. In addition to these traditional topics, the introduction of the concept of objects, the extension of modularity to packages, the principle of exception handling, and introducing reuse through generics, are all achievable at an earlier point in the curriculum if Ada is the first language taught.

The dilemma is a mirror of the opportunities: should such things as packages, tasks, and generics be introduced at an early stage in the teaching of computer science? Some educators feel that the application of the fundamental concepts of structured programming, the sequence, selection, and iterative structures, are enough to concentrate on during the first semester of language study. Others feel that more emphasis on formal methods, the proving of correctness, should pervade a curriculum that is now nearly devoid of such topics.*

This attention to fundamentals, though sound pedagogically, would tend to reinforce the tendency of the undergraduate computer science curriculum to reflect the techniques and tools of programming-in-the-small, and thus teach the unrealistic world-view that most software engineering curricula later have to correct. For instance, discussions of modularity at the lower levels of the curriculum still concentrate on its use as a means of decomposition or of avoiding "repeating code" a la subroutines, without introducing the ideas of coupling and cohesion which are so important to the verification and reuse of software parts. Therefore, it is worth considering the inclusion of basic software engineering principles in the early computer science curriculum, with Ada as the vehicle. This

[tm] Ada is a registered trademark of the U S. Government, Ada Joint Program Office (AJPO)

* David Gries is the leading American advocate of this position His *Science of Programming* (Springer, 1981) and Shaw, et al's *Fundamental Structures of Computer Science* (Addison-Wesley, 1980), which is used in teaching beginning programmers at Carnegie Mellon, both contain information on mathematical methods absent from the majority of introductory texts

means that we must face the fact that the traditional one-semester introduction to both programming and a language may not be enough time. Perhaps it was in the Fortran days when all you needed to know could be taught in that amount of time, but we know a lot more now, and we need more time to present it.

Some institutions have been taking the time. For example, The Wichita State University for nearly a decade has required a two-credit-hour course *Introduction to Programming* that precedes language study. The topics in that course include fundamental structures, modularity, and documentation techniques often left to much later in the curriculum. The results of this emphasis appear to be excellent, in that potential majors get an early introduction to the facts that programming does not equal hacking, and that mathematics and English are important, allowing the later courses to make use of extensive documentation techniques and continue the software engineering approach. In fact, Ada is one of the languages a student can study *immediately afterward*.

Still, for some time in the future in most schools Ada is likely to appear much later in the curriculum. Again a dichotomy presents itself: should we teach Ada in context of software engineering, or Ada as a vehicle to teach software engineering? In the last two years I taught courses with both approaches. The remainder of this paper contains "lessons learned" that hopefully provide insight into the dangers and rewards of each route.

Ada as a Tool in Teaching Software Engineering

It seems logical that a software engineering course should use good tools. Since Ada is considered the best programming tool for many domains, then Ada should be used as the language of choice when teaching the subject of software engineering. In the fall of 1986 I used Ada for the first time in an offering of an introduction to software engineering course aimed at seniors and first-year graduate students at Carnegie Mellon University. The full story of the course as a case study is in Tomayko87. Here we concentrate on the effects of using Ada.

The project chosen for the course was to develop a mission planning and simulation tool for a manned research station on the planet Mars. Fundamentally, the simulator was a batch run that calculated resource usage, equipment scheduling, and crew duties for different mixes of time, personnel, and materiel. System analysis, requirements definition, specification, and design using an object-oriented approach were completed prior to turning the design over to two teams: an Ada implementation team of five persons and a single Pascal coder. The high level and detailed design used an Ada program design language, and both teams very quickly created code. Quantitatively, the result was 16,000 lines of Ada and 8,000 lines of Pascal.

Since the object of the course was to teach software engineering, there was no formal Ada instruction. The class was divided up into requirements analysis, design, code, configuration management, quality assurance, management, and verification teams early in the semester and specialized in their fields. Therefore, the Ada coders had nearly two months for "training" prior to receiving the design for coding. During that time they worked on a graduated series of problems to hone their skills. They used no one text or training course, but succeeded in learning the language well.

The addition of two "language lawyers" helped enormously. These were a pair of European visiting students that became available after the semester began. They each had a year's Ada programming experience and influenced the coding team's work to a significant extent. In fact, during the class' final presentation of the project, given at the Software Engineering Institute, several experienced Ada practioners remarked on its quality.

The design team used an object-oriented method, based on research they had done in the available journal literature. Their decision to use Ada-like syntax to express the design accelerated the coding process tremendously. The use of the method resulted in an easily understood separation of function, high cohesion, and minimal coupling, which aided in testing the code later.

There were several lessons learned from using Ada in this course. One is that good programmers can learn to write good Ada fairly quickly. Packaging and generic concepts, limited and private types, all seemed to be internalized quite well. Tasking was not needed, so no information about the use of that construct was gained. Object oriented methods appeared to work sufficiently well for a system of this size. And again, students picked this up quickly.

The disadvantages of using Ada in the course arose from the development environment. The early coding was done on MicroVax workstations which were shared with other members of the Computer Science Department doing *experimental changes to the operating system.* This led to an unstable environment that often made the compiler unavailable. The compilations were very slow in execution. It took an average of an hour to recompile the developed system. An opportunity arose to move the system to a Vax 8600 and a different compiler, and when this was done and recompilation started we found numerous inconsistencies between the two compilers and spent a lot of time correcting "errors" that simply did not exist under the original system.

In general, the use of Ada in this case was a positive experience. The students applied their classroom knowledge of software engineering to a realistic project and found Ada to be a useful vehicle for implementation. The Pascal version of the code did not perform any better, and was much more difficult to read.

Software Engineering as a Tool in Teaching Ada

In the fall of 1987, I taught a course entitled *Ada and Software Engineering* which has been in The Wichita State University curriculum for several years. Whereas the purpose of the course described above is to teach software engineering and demonstrate Ada's use as a tool in implementing software engineering principles, this course has the objective of teaching how Ada directly supports the discipline of software engineering. The students in the former course are generally either inexperienced or modestly experienced. Those in the latter course are largely graduate students who have actual software engineering experience and are making, or want to make, the transition to Ada.

This course made use of a variety of readings and projects. Booch86 was the primary text, though not read in order, and several supplementary readings on environments and the design of real time systems using Ada were given. Individual projects included a design and implementation of software to control a fairly complex vending machine, a simple line-oriented text editor to demonstrate access types, and a task-based system to monitor and control a rocket engine. But I wanted to also demonstrate the use of Ada in a larger project, so again the introduction to software engineering course provided one of sufficient scope.

I was teaching both the Ada course and the software engineering course at the same time, so I decided to have the software engineering students do the requirements analysis, specification, high level design and verification of a large chunk of software, and have the Ada class do the detailed design using an Ada program design language and the code itself. The project was a simulation of command computer software for an unmanned, autonomous Mars rover.

The results of this experiment provided some new insights. It was much better to present Ada in the context of software engineering than to teach Ada in a beginning class or strictly as a tool. The emphasis on showing exactly how Ada is designed to implement data abstraction, principles associated with modularity, and other concepts made it quite easy for the students to understand the proper use of the language. When the time came to do the large project, I divided the students into two competing teams (named Deimos and Phobos, of course) to implement the design given them by the other class. The report of the verification and validation team showed that the two teams had nearly identical performance, and that 40 per cent of all the requirements were correctly implemented. This figure is amazing considering that the software engineering class delivered what was quite frankly a poor design, forcing the Ada class to go back to the requirements level to make sure that their detailed design included what was needed. Also, they had less than three weeks for detailed design, code, and unit and integration test.

One thing that really stood out in this experience is that the persons involved in design of an Ada program should have knowledge of Ada. The high level design delivered to the Ada class used primitive applications of software engineering principles simply because the designers could not imagine anything else. The Ada class "bootlegged" use of data abstraction and other principles after they saw how poor the design was --- a situation common in "real life" but by no means optimal.

Conclusion

My experience with these courses and in discussions of how best to teach and use Ada in academic environments lead me to firmly believe that Ada and software engineering are inseparable. Ada should be taught in the context of software engineering and the concept of the "programmer" should be dead, if it is not functionally dead already. The days of "Fortran for Engineers" courses must end. One of the students in the *Ada and Software Engineering* course was discussing his work at a local power company with me after the semester and said that he was trying to convince his co-workers to start a movement toward Ada because he "couldn't think in any other language any more." I feel that what he was really saying is that he can not think as a programmer any more, that he now thinks like a software engineer and he knows that the best implementation of software engineering is Ada.

Acknowledgement

My knowledge of Ada was gained most effectively by my participation with my students in these classes and in other learning situations. I want to thank them for their active interest and their difficult questions.

References

[Booch86] Grady Booch, *Ada and Software Engineering*, 2nd Edition (Benjamin Cummings, 1986).

[Booch87] --- *Software Components with Ada* (Benjamin Cummings, 1987).

[Tomayko87] James E. Tomayko, *Teaching a Project-Intensive Introduction to Software Engineering*, Software Engineering Institute Technical Report SEI-TR-20, October, 1987.

James E. Tomayko is an Associate Professor and Director of the Software Engineering Program in The Wichita State University Computer Science Department. He spent 1986-1987 as a Senior Computer Scientist at the Software Engineering Institute. Dr. Tomayko has done contract work for NASA and the Boeing Military Airplane Company.

CHANGING THE STUDENT'S PERCEPTION OF SOFTWARE LIFE CYCLE

M. Susan Richman, Ph.D.

The Pennsylvania State University at Harrisburg

Abstract

The life cycle of a student's program written in a typical programming course consists of the following stages: (1) write the program, (2) debug the code, (3) execute the program, (4) turn in for grading, (5) glance at the graded program, and (6) discard.

This paper describes the experiences of one instructor in trying to instill, through programming assignments, an appreciation of the value of designing and coding for understandability, modifiability, reusability, and reliability. Thanks to various features of Ada, especially packages and the facility for the separate compilation of program units, the student can experience, in the confines of a one-semester course, some of the concerns and difficulties of members of a development team and maintenance programmers.

Introduction

In many introductory programming courses, the only reason a student modifies code is in order to debug it. Often the student has had no experience in trying to understand or maintain code written by someone else, or by himself or herself after several months have elapsed and the familiarity has faded. Furthermore, the code written is seldom useful enough to be used in more than one program. As such, although the instructor may stress the importance of the principles of software engineering, the student generally has little practical appreciation for the benefits derived and has an artificial and misleading perception of the software life cycle.

At the same time, the instructor may have difficulty in designing programming projects that are of significant size and yet might reasonably be assigned to the students who, the instructor is not permitted to forget, typically have the demands of four or five additional courses placed upon them during that same semester. Ada, by providing the modularity through which large systems are build from simpler software components, assists in coping with both these difficulties.

The student, starting very early in the course, is required to build software components which later are used, not only in one larger system, but in several systems. Students modify their own code to meet new requirements, and they modify other students' code (and critique the code they must maintain.) They applaud readable and understandable code, and are appropriately frustrated by someone else's AdaTRAN which they must decipher in the course of this assignment. Through modifying code, they also recognize the value of modular program units that diminish the ripple effect of changes. In the course of enduring these experiences, during which students become exasperated by poorly designed code and are correspondingly gratified when able to work with well designed systems, they develop a new appreciation for the value of Ada's support for software engineering and a more realistic view of the software life cycle.

Basic Course Information

The course, COMP 408, Introduction to the Ada Programming Language, is taught at Penn State, Harrisburg, an upper-division and graduate school. As such, there are no freshman or sophomores in the course. COMP 408 is a three-credit, one-semester course with a prerequisite of at least three credits of high-level programming language. Most students taking the course have previously studied Pascal. The computer used is a Data General Eclipse MV/10000 and the compiler is the Data General/Rolm compiler with the Ada Development Environment.

Programming Assignments

==> The first programming assignment is designed to assist the student in becoming familiar with the system, the Ada environment, and the basic structure of an Ada program. Through part 4 of the assignment, they learn how to read from, and send output to files.

Assignment 1
CHECK PARENTHESES PROGRAMS

1. Following the instructions in the "Beginning ADE Users' Guide" create a Program Root Directory. In this directory, or in a sub-directory, if you wish, type the procedure Check_Parentheses listed in 1-2 of the Class Notes. Compile, link, and execute this program. [The input for this procedure is to be entered from the terminal.]

2. Compile, link, and execute procedure Match_ Parentheses in 1-4 of the Class Notes. The input for this procedure is to be entered from the terminal.

3. Compile the package Parentheses in 1-9 of the Class Notes. Write a program (as a main procedure) to utilize this package to accomplish the actions of procedure Match_Parentheses.

This program should accomplish the results by following the general form of Match_Parentheses but calling the facilities of the package:

```
      procedure Tally_Unmatched Left
      function Have_Unmatched_Right
      function Have_Unmatched_Left
```

in place of the appropriate algorithms within the body of Match_Parentheses. Compile, link, and execute this procedure.

4. Match_Parentheses received its input from the keyboard and outputs the result to the terminal screen. The procedure in Class Notes (1-3) uses Ada statements to create and send output to a file. Use this procedure to create a file called INPUT.DAT. After creating this file, use an editor to insert text in this file. Be sure to include an assortment of left-parentheses and right-parentheses in this text so that you can use your Ada program to check for matching left- and right-parentheses. Be careful not to interfere with the End_Of File mark that the Ada program has placed.

The procedure in Class Notes 1-5 shows how to read characters from an Input File rather than from the keyboard.

Modify your procedure from Part 3 (above) to read the text from your file INPUT.DAT and check for matched or unmatched parentheses.

==> With the second assignment they begin to write and to use separately compiled library units.

Assignment 2
FACTORIAL TABLE PROGRAM

Write an Ada procedure that prints a table of factorials for the integers from 0 to 10. You will need a function Factorial that computes N! Write it as a separately compiled library unit. Using the fact that N! = N*(N-1)!, write Factorial as a recursive function. The specification of Factorial will look like:

```
 function Factorial (N : Natural) return Natural:
```

Your output should look similar to:

```
     n         Factorial (n)

     0         1
     1         1
     2         2
     3         6
     4         24
          etc.
```

[15 spaces between the columns]

Next modify this program with the following changes:

(1) Use the integers 0-5 instead of 0-10;
(2) Put 10 blank spaces between the numbers.

You are encouraged to make minor changes in your original program, if necessary, so that your program would be easily modifiable. Aim for easily modifiable programs by using appropriate Ada features.

==> Assignment 3 modifies the procedure from Assignment 2 and accesses a function provided by a system library, Math Lib.

Assignment 3
SQUARE_ROOT TABLE USING MATH_LIB

Modify your Factorial Program to print a table of square roots of the floating point equivalents of the first 10 integers, using the function Square Root found in the Ada Development Environment package Math_Lib.

In order to use this package containing mathematical subprograms you will need to change the Library Search List for your project Root Directory to include the Math Library. The instructions for doing this are found on page 18 of Beginning ADE User's Guide.

==> Once composite data types are introduced, library packages are designed to define and perform I/O on them.

Assignment 4
ARRAY_IO PACKAGE

Write a package providing the facilities [Get and Put] you need to perform Input/Output on an array of integer components. For the procedure Get, the components will be read one at a time, and for the procedure Put, written one at a time, in tabular form with the index value in the left column and the corresponding component value in the right column.

Give attention to making the procedures easily modifiable. [This won't be the last time you will see your package.]

RECORD IO PACKAGE

Write a package providing the facilities needed to maintain records for a car dealership. This package should include:

(1)A type definition for Car_Record_Type which contains information such as:

```
     Make  (choose just a few possibilities):
     Year
     Miles_Per_Gallon    --a floating point type
     Miles_travelled     --an integer type
     Price               --a fixed point type
```

(2) A procedure to Get a car record:
(3) A procedure to Put a car record.

Incorporate File_IO to Get input from a file (previously created using the editor) and to Put output to a file created by the Ada program.

****

WHILE YOU SHOULD TEST THESE PACKAGES WITH DRIVERS [FOR YOUR OWN SECURITY], THESE ASSIGNMENTS NEED NOT BE TURNED IN AT THIS TIME. YOU WILL NEED THE PACKAGE ARRAY IO FOR THE NEXT ASSIGNMENT AND THE ARRAY_IO CODE SHOULD BE TURNED IN WITH THAT PROGRAM. RECORD_IO WILL BE USED LATER.

==>The next assignment involves modifying Factorial to incorporate arrays, writing a new procedure Fibonacci, enclosing both procedures in a library package, and writing a driver to use this package and Array_IO.

Assignment 5
FIBONACCI FACTORIAL PROGRAMS

Write an Ada procedure Fibonacci that computes and stores in an array the sequence of the first N Fibonacci numbers. The value of N should be an in parameter and the value of the array an out parameter. The value of N should be input from the terminal to the driver. The sequence of Fibonacci numbers follows the pattern

1, 1, 2, 3, 5, 8, 13 ...

so that, after the first two Fibonacci numbers, each is the sum of the two previous values.

Redesign the function Factorial into a procedure that computes and stores in an array the sequence of the first N factorials. The value of N should be an in parameter and the value of the array an out parameter. The value of N should be input from the terminal to the driver. Do not have the procedure Factorial use the previously written recursive function Factorial. Compute each factorial value using the previously stored value in the array.

Write a package which incorporates both procedures Fibonacci and Factorial. Write and separately compile a driver which uses these procedures and then the Array_IO to output the arrays.

==> A Person_Package is designed to deal with a variant record.

Assignment 6
PERSON RECORD IO
[DISCRIMINATED RECORD]

Write a Person_Record_Package providing the facilities needed to maintain records for a hospital. This package should include:

(1)A type definition for Person_Record_Type [a discriminated record with discriminant Max_Length of subtype positive, default value 20] which contains information such as:

Surname (a string length Max_Length)
Length of Surname (actual length)
Birthday (a record type)
Social Security Number (an array of digits)
Insurance Carrier (an enumerated type)

(2) A procedure to Get a person record:

(3) A procedure to Put a person record.

Incorporate File_IO to get input from a file (previously created using the editor) and to put output to a file created by the Ada program.

Compile the package body separately from the package specification. Write the procedure bodies of Get and Put as subunits, compiling them separately from the package.

(1) Two procedures to Get a person--one from the keyboard and one from a file; [Use Get_Line to read the person's surname.]
(3) Two procedures to Put a person--one to the screen and one to a file.

Compile the package body separately from the package specification. Write the procedure bodies of Get and Put as subunits, compiling them separately from the package.

==> The next assignment creates a more elaborate structure of car records, a linked list.

Assignment 7
INVENTORY OF AUTO RECORDS
[LINKED LIST]

Implement a linked list of Car_Type to maintain an auto inventory. Each cell in the linked list should contain a data item of Car_Type as well as an access value pointing to the next cell in the list.

Use Car_Package for the definition of the Car_Type and for the IO facilities. The linked list package should contain:
(1) a List_Type containing two variable values, Head, and Tail, indicating the beginning and end of the list.

(2) a procedure Insert to add a Car record to the list, in order by Make,

(3) a procedure Remove to delete a Car_Cell from the list

(4) a function Is_Empty to report when there are no records in the list.

(5) a procedure to Write the contents of the list (without the access values) to an output file. Car records should be read from a file.

Compile the package specification and the package body separately. Write the bodies of Insert, Remove, and Found as subunits and compile them separately.

An appropriate exception should be raised (and handled) upon an attempt to Remove from an empty list.

Use Unchecked_Deallocation to release the memory used for each element after it has been Removed.

Write a driver to test the facilities of the package.

==> The next assignment provides the facility to search through the linked list in Assignment 7. Each student modifies a classmate's code as well as his or her own.

Assignment 8
SEARCH THROUGH LINKED LIST
[MODIFY CLASSMATE'S CODE]

This assignment will give you experience in maintaining code written by someone else. You will be assigned a classmate with whom you will exchange the code from Assignment 7. This code is to be modified as described below. Make the modifications described below on your classmate's code and on your own. Compare and contrast the experiences.

(1) Incorporate in the package Linked_List a function Found which searches through the array for a given car record (passed to Found as a parameter) and returns the value True if it is found. The function will compare the given record only with the data item of each cell, not the access value(s) into account when testing for equality.

(2) Modify the procedure Insert so that a new cell is inserted only after function Found indicates that the data item is not already listed in the inventory.

(6) a function More_Efficient to compare two Car records and return the one with the greater number of Miles_Per_Gallon,

(7) a function Most_Efficient to search through the list and, using More_Efficient, return the Car with the highest Miles_Per_Gallon rating.

Write a driver to test these facilities.

The final project is to write a generic linked list, based upon the results of the previous two assignments.

FINAL PROJECT ASSIGNMENT
GENERIC LINKED LIST

Modify your List_Package to acquire a generic package providing dynamically allocated linked list structures.
This Generic_List_Package should provide:

(a) a generic parameter, Element_Type which will play the role of Car_Type

(b) a generic parameter, function Better to compare two Elements and return the one judged to be the Better of the two,

(c) an access type which accesses objects of Element_Type

(d) a private type List_Type containing a pointer to the Head of the List

(e) a function Is_Empty to return the value True if the list is empty

(f) a function Found which searches through the list for a given element (passed to Found as a parameter) and returns the value True if the element is found. The function will compare the given element only with the data item of each cell; it will not take the access value(s) into account when testing for equality.

(h) a function "<" to compare two elements and return the Boolean value True if the first is less than the second,

(i) a procedure Insert to insert an Element in the list, after using a "<" to locate the appropriate position in the list

(j) a function Best to search through the list and, using a generic parameter function Better, return the Element judged to be the Best of the list,

(k) a procedure Remove to remove an element from the list.

(l) a procedure to Write the contents of the list (without the access values) to an output file.

After compiling the generic list package, modify your driver for the Car list package to instantiate the generic list package for Car records. The driver should then call upon the facilities of the list package (the instantiation, not the generic package) to create a list, to insert the records in order by Make after the driver reads the records from an input file, to remove a record from the list, to search through the list and return the car with the highest Miles_Per_Gallon, and to write the contents to an output file.

M. Susan Richman has been teaching Ada at the Pennsylvania State University, Harrisburg, since 1984 and has been Director of the Ada Education Center at Penn State Harrisburg since 1985. She is a graduate of the University of California, Berkeley, and received the Ph.D. in Mathematics from the University of Aberdeen, Scotland.

M. Susan Richman, Director
Ada Education Center
Penn State Harrisburg
Middletown, PA 17057
Phone: (717)948-6082

Examination of Some non-Ada Software from the Pespective of Software Engineering Principles, Processes, and Goals.

by

Jagdish C. Agrawal
Professor and Chairman,

Embry-Riddle Aeronautical University,
Computer Science Department
Daytona Beach, FL 32014

A number of software products were examined from the perspective of Software Engineering Principles, Processes, and Goals (how well the principles were implemented, what specific processes and tools were used, and how far the goals were achieved). An attempt was made to also examine whether the use of Ada would have made the difference. For some software products, the language did seem to make the difference. For example, COBOL allowed little or no information hiding, and left everything quite visible.

1. Introduction

The so called software crisis is characterized by the issues of cost, efficiency, modifiability, reliability, transportability, and understandability. Even if the software system meets its initial functional requirements, its efficiency, modifiability, reliability, transportability, and understandability can have a major impact on its usability and its life-cycle cost. Therefore, it may be useful to study some of the existing software from the perspective of characteristics it possesses from the above list, and the factors that contributed to missing the characteristics it does not possess. This belief became the motivation for this research.

The approach for the research, described in section 3, was to examine each of the selected software and identify what software engineering processes were used, what software engineering principles were followed and implemented, and what software engineering goals were actually achieved. An attempt was made to identify whether the use of Ada would have made a difference.

For this research, a model of the Process, Principles and Goals (PPG model) of Software Engineering was needed. I started with the PPG model of Ross, Goodenough and Irvine [1] based on their thesis that "Four properties that are sufficiently general to be accepted as goals for the entire discipline of software engineering are modifiability, efficiency, reliability, and understandability." Recently, to these four goals, Agrawal and Manickam [2] added a fifth goal of portability. Thus, for the study of several software products, I slightly modified the Ross, Goodenough, and Irvine model for the Process, Principles, and Goals of Software Engineering [1]. This modified PPG-model is described in the next section.

2. Modified PPG Model

There are five goals in the modified PPG model used for this work.

- **Modifiability**
- **Efficiency**
- **Reliability**
- **Understandability**
- **Portability**

The first four in the above list are defined and discussed in detail by Booch [2, pp. 28-31]. Rapid technological developments and sharp drop in the cost of hardware has motivated users to upgrade and proliferation in their hardware and support environments. This has made portability an important goal. Software portability across various hardware and support environments also has an impact on reducing the time and cost of learning curve for the staff. We can call this reduction in the time and cost of learning curve for the staff as "staff portability." For this research, we viewed both the software portability and the staff portability important enough to include the general term portability in the list of our software engineering goals.

The seven principles of software engineering proposed by Ross, Goodenough, and Irvine [1] were adopted without modification:

Abstraction: Extracting essential properties while omitting inessential details.

Completeness: Ensuring that all of the important details are present.

Confirmability: Explicit statement of the information needed to verify correctness.

Hiding: Making inessential information inaccessible.

Modularity: Purposeful structuring into relatively independent parts to easily achieve some purpose.

Localization: Bringing related things together into physical proximity.

Uniformity: Using consistent notation among all modules to directly support the goal of understandability.

For the software engineering processes in the PPG-model used, we adopted the following version of the five processes used in the Ross, Goodenough and Irvine model:

Purpose: The requirements of a system.

Concept: The architecture of a software system to satisfy these requirements and specify the modules that constitute the system.

Mechanism: The hardware and support environment in which the software runs.

Notation: The programming language for implementation of the software.

Usage: How the software system is controlled (e.g., User's manual).

3. Evaluation of the Software against the modified PPG Model

A large research team participated in this project. Senior and advanced junior students in my Software Engineering class were first given the Software Engineering background equivalent to chapters one through nine, and chapters 22 through 24 of Software Engineering with Ada by Grady Booch [1] and also introduced to "Software Engineering: Process, Principles, and Goals," by Ross, Goodenough, and Irvine [1]. They were also asked to read a number of books and papers on this subject that were placed in the reserve section of the university library. Then they were asked to individually select a software of their choice and examine it from the perspective of my modified PPG model of software engineering. Technical research reports of the students summarizing their findings were critically examined by me and I further examined each software for two issues:

a. Goals that were not achieved, and

b. Will the use of Ada have made any difference.

The material used for the evaluation included source code where available, documentation, information on system change requests and results where available, interviews with users and developers, and available documentation on documented complaints with the system. Not all of these were available for each software available, and therefore the evaluation lacks uniformity and consistency. However, we believe that studies of this type are very valuable for large system developers.

4. Summary of Results

Results are summarized below separately for each software considered. For brevity in the paper, I have omitted the details about the principles that were followed and the goals that were met in each software. Only the shortcomings are pointed out. Also omitted are the specific details that were necessary to arrive at the conclusion that a principle was followed or not followed, and a goal was met or not met. The documentation on such matter is quite bulky to reproduce and is now the property of Embry-Riddle Aeronautical University.

(i) Applications Software for Creating Shapes

Purpose, concept, mechanism, and usage is identified in related documentation [4, pp. 206-234].

Notation: Applesoft BASIC.

Principles not implemented with reasons:

Hiding: Since the language of implementation does not support hiding, the principle of hiding was not and could not be followed. Use of Ada certainly would have helped in this area.

Goals not met with reasons:

While relative independence of the modules, and strong documentation on purpose, concept, mechanism, notation, and usage made the software easily modifiable with regard to new requirements, the portability was severely restricted by the mechanism and notation used. If the requirements on the mechanism changed to include hardware other than a 6502 processor and resulted in change in the operating system, then portability will become very important. Uniform language standard of Ada regardless of the hardware is an important feature that helps portability of at least the source code.

(ii) Bulletin Board System (BBS)

Notation: BASIC and Assembler

Mechanism: Atari

Purpose and Concept: The Bulletin Board System BBS was developed by Optimize Systems, Inc. for telecommunications use of Atari Computer users. Source code was not available for this proprietory product.

The developer claimed having implemented all the principles of software engineering. However, in my opinion, conformability was substituted with hybrid prototyping by using controlled user feedback for about six months on initial versions of the software to develop the first commercial version. Also, uniformity was some what lacking resulting from a mixture of Assembler with BASIC.

The goal of portability was difficult to enforce, because the user did not have the source code. Even for the developer, to port the software to a different hardware and support environment would have resulted into several man-months of effort. Use of Ada would have helped the portability at the source code level, but the machine dependent features of BBS would have restricted the ability of even Ada to produce a portable product.

(iii) Test scorer program

This is an in-house program developed to satisfy the needs of upto about 300 instructors who could use it for machine scoring of their class tests. In a university environment, such programs improve the productivity of a professor and free up some of his time for research that would otherwise have gone into manual grading of the tests. It has gone through several System Change Requests over the last several years of its use. Thus modifiability has abundantly been demonstrated.

Notation: Turbo Pascal

Mechanism: IBM PC and scan-tron machine

The modules were not entirely independent of each other, and control through GOTOs and interrupts made the implementation of modularity weak. The principle of hiding was not practiced and the language of implementation made it somewhat difficult to enforce hiding. The GOTO dependent design/implementation compromised any efforts to achieve localization.

The goals of software engineering were achieved although the source code would still require reimplementation effort before porting it to another environment.

Use of Ada for initial implementation is unlikely to have produced any better or any worse results.

(iv) LAB ASSIST

This is an in-house developed program for scheduling of laboratory hours and assistants.

Mechanism: IBM PC and PC DOS
Notation: BASIC

There is a high interdependence among modules and it is strongly recommended that at the redesign time, developer ensure that only the interfaces of the modules be defined and visible. Abstraction is lacking and for any redesign effort, it is recommended that the redesign place a high value on abstraction. Localization is also lacking. The goals of modifiability and portability will benefit from a major redesign as recommended earlier.

Implementation of the redesign in Ada will certainly assist in attaining all the goals. A major redesign with possible implementation in Ada is in process.

(v) Grading Program

Much like the Test Scorer Program, this too is an in-house developed program with the functional requirements somewhat different from that of the Test Scorer Program.

Mechanism: PRIME 400 (later ported to IBM PC with Turbo Pascal)

Notation: Pascal

All the principles were reasonably followed.

Lack of detailed documentation became an obstacle to achieving modifiability. Source code availability did not help much in the requirements of porting from PRIME 400 / Pascal to IBM PC / Turbo Pascal. Use of Ada would have certainly helped in both. This seems to be a case where a simple translation into Ada would be acceptable until such time that huge changes in the requirements make a major redesign necessary.

(vi) EASYNAV

The system EASYNAV is a flight management system that was designed in-house for use by pilots in performing preflight computations. It is a first step prototype towards a real-time embedded automatic flight control system which would require porting the software system to a system different from the one on which it has been developed.

This is another example where we felt that the tasking capabilities of Ada for its real-time embedded system use will make it an excellent candidate for implementation in Ada.

(vii) Propeller Performance Evaluation System

Inhouse developed software for use in our Aeronautical Engineering Department to evaluate propeller performance and loading in uniform and non-uniform fields.

Mechanism: PRIME 400

Notation: FORTRAN 77

The software is intended for use by aerospace engineers who are working on the performance of propeller flowfields.

All the principles were followed and goals achieved. Use of Ada is unlikely to bring any new benefits to the current implementation.

(viii) Data Item Attribute Load (DIAL)

The program was developed in 1986 for use by a large aircraft company.

Followed all principles and met all goals. Use of Ada is unlikely to improve anything.

(ix) Accounting Partner II

Product of Star Software Systems.

Notation: Compiler BASIC CB-86

Mechanism: 16-bit microcomputer that supports CB-86

The manuals are clear, concise and truly user-friendly.

Modularity was weak in the system, primarily due to the limitations of the language of implementation. CB-86 also made hiding, abstraction, and localization difficult to achieve. As a result, the goal of modifiability was not fully achieved. Portability of this software will be substantially enhanced by Ada. Efficiency was traded off for file retrieving capabilities.

(x) Automatic Acquisition System

This is a Columbia University developed system that provides sophisticated records and control for all in-process and fiscal functions of library acquisitions. It is operating with efficiency and economy and satisfies the goals and principles of software engineering. Implementation of the system in Ada would certainly enhance portability.

(xi) Interior Designer System

The system was developed by a commercial developer for an Interior Designer Company in Florida.

Notation: Turbo Pascal

Mechanism: IBM PC with Graphics

Intended for use by an interior designer for simulated prototyping of interior design and its enhancement with the customer participation.

Although the system did not entirely follow the principles of abstraction, hiding, modularity, and localization and did not completely achieve the goals of portability, efficiency, and and modifiability, it is important to note that the user was happy with the system. Implementation in Ada would have had a high front end cost that user perhaps would not have considered reasonable.

(xii) Institutional Advancement System

This is a proprietory, in-house developed system for use by the Embry-Riddle Aeronautical University's University Relations Department. The use of the software is related to the fund-raising activities of the university.

The software engineering principles were followed, although a large turnover of the developer's staff during the development phases made the documentation insufficient and understandability weak. Modular independence through Ada's packaging concept would have overcome this weakness.

(xiii) Structural Analysis Program (STRUXR)

This system was developed under the leadership of Aeronautical Engineering Professor, Dr. Howard Curtis for Structural Analysis. It required one-man year effort for the design and implementation phase after the requirements and specifications were developed on the basis of several years of experience by Professor Curtis.

Notation: FORTRAN

Mechanism: HP 1000

There are thirty five modules relatively independent of each other. The only principle not followed is hiding and that is because the language of implementation does not permit one to. The goal not achieved well is portability and a major effort to port the system to FORTRAN 77 in the IBM 4361's IX-370 environment (IBM's Unix System V, hosted within VM) is currently in progress. Unfortunately, we have not acquired Ada on IBM 4361 because of its cost, and that has been the major reason why a redesign and reimplementation in Ada has not been attempted.

(xiv) FASTRAC

FASTRAC is a copyrighted system created by Dyer Wells & Associates in 1985 and adopted and installed by Special Agents Mutual Benefit Association (SAMBA) in 1986. It is used to process claims for medical, dental, disability, major-medical benefits etc., generation of checks, reports, and keeping a record of claims processed etc.

One of the student researchers was able to examine some documentation on the software system to make the evaluations. COBOL was used as implementation language.

All principles were followed to varying degrees. However, the system did not achieve the goal of modifiability satisfactorily. This was primarily due to the fact that COBOL does not permit hiding and as a result modules are not totally independent. A small change in one location tends to have an impact at several other places within several modules. Use of Ada would certainly help in such an application.

(xv) Student Evaluation Software

This is an in-house developed system to analyze student evaluations of their instructors and courses and producing summaries of the analyzed data.

Notation: Turbo Pascal

Mechanism: IBM PC

This is a relatively small applications software. All the principles were followed and all the goals were achieved, including the goal of portability. Unless reuse of this software was planned in other applications, reimplementation of it in Ada does not make sense. Even in case of planned reuse, one could use pragma interfaces to this software, rather than absorbing high expense of reimplementation of this software in Ada.

(xvi) Data Entry Keyboard Indicator (DEKI)

The purpose is to enter data and convert discrete information for transfer to the F-14D and A-6F aircraft. Communication with the aircraft system is via a dual standby, multiplex MIL-STD-1553 Data Bus. DEKI assumes the role of a remote terminal. The mechanism includes a 20 push-button scratchpad and a 80186 processor. While the principles were practiced and goals achieved to a large degree, implementation in Ada is recommended for the type of use that may need portability across multiple hardware and support environments.

(xvii) Math and English Placement Test Software

This is an in-house developed software for grading, and analyzing Math and English tests of incoming freshmen and for report generation. It has been successfully used since 1981.

Notation: COBOL

Mechanism: HP 3000, HP 2621 terminal, and SCAN-TRON 2012.

A hardware upgrade in 1986 from HP 3000 to IBM 4361 placed requirements for porting the software. The porting effort has been large and not entirely a grand success. This is enough to say that the language of implementation initially was not ideal for portability and the software did not meet the goal of portability. Confirmability principle was not strictly enforced.

If the university decides to continue using English and Math Placement Exams, my recommendation will be to redesign this software and reimplement it in Ada, so that future hardware upgrades will not impact it as harshly as was the case recently.

(xviii) CADcompiler

This system was designed by a group of design engineers for use on a personal computer. All principles were followed and goals met. Not enough information was available to make any judgments on desirability of reimplementation in Ada.

(xix) Labor Tracking System

This software was developed by a large aircraft company and currently in use at a large corporation's Simulation and Control Systems Department.

Notation: FORTRAN (somewhat antiquated version)

Most of the software engineering principles were followed well, yet not all goals are achieved. The language of implementation and software complexity introduced through revisions done in the maintenance phase have made modifiability, understandability and portability very weak. This seems like an ideal candidate for a major redesign and implementation in Ada.

(xx) Medical Management System

This system is used by medical agencies for billing Medicare, HMO's, and other private insurance companies.

Notation: COBOL

Mechanism: Microcomputers that support COBOL.

The system performs book keeping, accounting, and record keeping functions, and provides comprehensive statistics, electronic billing, etc.

Although the system is modularized, there is little localization, hiding and abstraction. Portability and Modifiability can be improved with Ada.

(xxi) Radar System for the F-16 Flight Simulator

Designed by a large corporation under a government contract for Air Force, this system is serves to display to a training F-16 pilot the various patterns of the area he will be flying over for a particular mission, and familiarizes the pilot with all necessary procedures to be carried out prior to the real flight. It is useful in reducing pilot error, and cost.

Notation: 450,000+ lines of FORTRAN 77

All principles were practiced, although it was difficult to enforce hiding and localization with the limitations of FORTRAN 77. While the goals were achieved to varying degrees, use of Ada will certainly enhance modifiability and portability. Understandability of each module will also increase due to stricter localization.

5. Conclusions

Initially, the project started out with forty five participant student researchers, each of whom was to find a software product of his/her choice for evaluation. About half of them dropped out, perhaps due to the severe time requirements in the project. As a result, only 25 software products were examined and evaluated. Of these, four did not apply all the evaluation criteria and they were not included in this paper. The paper summarizes the findings in twenty-one evaluations. It is clear that Ada may not be the solution for every software problem, as has been pointed out in some cases above from the points of view of cost, effort and use, in a majority of cases, whenever there is a redesign, and the principles of hiding, localization, and abstraction need to be enforced strictly, use of Ada should be seriously considered. Good use of Ada can also enhance portability and modifiability. Relative independence of modules and Ada can help enhance the understandability of each module.

Use of this methodology for evaluation of a large system against a PPG-model of software engineering before the system goes for a major maintenance or for a complete redesign is very useful because the results of the evaluation can produce useful recommendations for a redesign/maintenance effort. LAB ASSIST and MATH AND ENGLISH PLACEMENT TEST SOFTWARE discussed above, have benefited from the evaluation. Large software houses will enhance quality in redesigned products by evaluation of the product ready for redesign.

6. Acknowledgments

The entire research team that participated in the research project deserves part of the credit for this work. First, Professor Steve Glassman of the English Department read each report and made comments from the perspectives of a user/manager who may not be a software engineer but needs to know the results. Professor Glassman's contributions were found to be very valuable by me as well as by every member of my research team. All members of the research team also deserve acknowledgment and recognition for their contributions to this rather large project. These members are: Hassan AL-Mousawi, J. Alcaide, George Arroyo, Peter Bauert, Don Burke, Charles Carey, Andre Charron, Bryan Collier, Reginald Davis, Greg DiPenta, Danette Gracia-Moorhead, E. K. Guruvadoo, C. Malva Ketter, Jeffrey Limback, Raynard Logan, Richard Manuel, Emerick Martin, John Martin, Michael Panciroli, Richard Phinney, Margaret Poteat, Kirk Poulson, Patricia Renwick, Cary Rotter, Glen Shelley, Asoka Rabindra Singh, and Marcos Yanes. There were seventeen others who started the evaluation but either never finished it or did not submit a written report on the evaluation.

My sincere thanks and appreciation is due to Ms. Jane Diehl, my secretary, who placed a high priority on my manuscript and helped me meet the deadline.

7. References

1. D. T. Ross, J. B. Goodenough, and C. A. Irvine, "Software Engineering: Process, Principles, and Goals," Computer, May 1975.

2. J. C. Agrawal and S. Manickam, "Counting Leaves: An Evaluation of Ada, LISP, and PROLOG," Proceedings of the Fourth Symposium on Empirical Foundations of Information and Software Science, Plenum Publishing Company, New York, 1987.

3. G. Booch, Software Engineering with Ada, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

4. A. R. Harriger and J. C. Agrawal, Applesoft BASIC Programming, Merrill Publishing Company, Columbus, OH, 1987.

Jagdish C. Agrawal
Professor and Chairman
Computer Science Department
Embry-Riddle Aeronautical University
Daytona Beach, Florida 32014

Jagdish C. Agrawal is a professor and chairman of the Computer Science Department at Embry-Riddle Aeronautical University, Daytona Beach, Florida. His research interests include Software Engineering and Distributed Computer Systems. Prior to joining the Computer Science Division at Army Institute for Research in Management, Information and Computer Sciences (AIRMICS). He also chaired the Computer Science Technology Department at Purdue University, Ft. Wayne and held professorship in the Department of Mathematics and Computer Science at California University of Pennsylvania. He received his Ph.D. from Purdue in 1969, M.S. from University of Windsor in 1966, M.Sc. from Agra University and B.Sc. also from Agra University.

An Ada[1] Training Life Cycle Curriculum

Daniel J. Connolly

GTE Government Systems

1 Abstract

This paper describes the Ada software engineering methodology, and the Ada life-cycle training curriculum based upon that methodology that is being implemented at GTE Government Systems in Rockville, Maryland. The paper also discusses how decisions regarding the content, organization, timing, and method of presentation for the curriculum are made. Finally, it presents an assessment of the training curriculum to date.

2 Introduction

The Ada training curriculum described in this paper is an ambitious attempt to prepare over 170 software engineers with the necessary skills to develop large, (greater than 200,000 source lines of code) real-time, distributed systems in a DoD contracting environment. This software engineering organization was originally established to support a large project which was originally bid to use "C" as the implementation language. Early in 1986 during the software requirements analysis phase, a decision was made to use Ada as the implementation language. The lack of experienced Ada software engineers (less than fifteen percent of having greater than six months Ada experience), the use of a new methodology, and a new language within a new organization made it imperative that the training program be effective in assimilating software engineers into the development environment. This meant providing instruction not only in the Ada language, but also in the philosophy of Ada software development, the activities associated with the development effort, the products of software development, the standards for those products, and the software tool environment.

Early efforts in bringing in training vendors for Ada proved disappointing for three significant reasons.

A. Schedule - The training courses attempted, and failed, to prepare Ada developers within a short period of time, usually one or two weeks. This period was inadequate, considering the complexity of the Ada language, the need for practice time, and the lack of Ada experienced software engineers.

B. Content - The training courses were focused primarily on syntax and semantics, and did not suggest a software development methodology, nor were they sensitive to our in-house methodology. They were too general in content, so that they were inadequate for any particular audience, such as technical managers. They were designed more for ease of presentation, and less for promotion of learning.

C. Cost - The training courses were too expensive, and considering the limited benefit were not cost-effective.

To address these problems, and the needs of the software development organization, an Ada training curriculum was designed with the following characteristics. This curriculum:

A. Begins with an intensive, three week Ada language course, which covers the entire language and its features.

B. Includes a series of Ada software development methodology courses focusing on the transition between each of the major phases of Ada software development as defined in our software engineering methodology.

C. Includes a course specifically tailored to the needs of technical managers.

D. Emphasizes the principles of software engineering, and the philosophy of Ada software development.

E. Is cost-effective for training over 170 software engineers over two years.

F. The courses within the curriculum present information in a manner that promotes understanding by the student through careful organization and relevant examples and exercises.

In order to provide a context for discussion of the training curriculum, a discussion of the software engineering methodology will be presented. This will be followed by a discussion of each course in the curriculum, and an assessment of the training program as of December, 1987.

[1] Ada is a trademark of the U.S. Government (Ada Joint Program Office)

3 Methodology

The Ada software engineering methodology entails three areas: the phases of software development and their associated activities and products, the software component structure, and the documentation requirements.

3.1 Impact of Ada

The GTE Ada software development methodology is a refinement of a standard DoD life-cycle model. This model identifies five phases of software development: software requirements analysis, software preliminary design, software detailed design, software implementation, and software integration and test. In order to address Ada software development issues, refinement of certain life-cycle phases into subphases was necessary. Also, modifications were made to the standard products associated with each phase in order to accomodate Ada.

The DoD life-cycle model identifies a component structure which provides guidance for how the software development products are to be organized and managed. These components are: computer program, computer program component, unit, and routine. Ada development requires a modification to this component structure, and the mapping of the structure to the life-cycle phases.

The DoD standard requires certain deliverable documents for each phase of software development. The combination of the inapplicability of the component structure, the static, functional nature of the document organization, and the inevitable redundancy of information forced a revision to the standard documentation.

The traditional software life-cycle phases, based mainly on functional decomposition methods of software design, and a baseline-oriented approach towards documentation, are significantly incompatible with common approaches to Ada software development. Modifications made to the phases, activities, products, component structure, and documentation are discussed by phase.

3.1.1 Software Requirements Analysis Phase

The software requirements analysis phase focuses on analysis of system functional requirements to determine software requirements. This analysis was performed through structured analysis techniques as described by Ward and Mellor. Analysis products included dataflow diagrams, events lists, state-transition tables, entity-relationship diagrams, and data dictionary.

Software requirements were allocated to computer programs for management and documentation purposes.

Resulting from the analysis was a rigid description of the software requirements documented within Software Requirement Specifications, and data definitions documented in the Data Dictionary Document.

There was no change made to this phase as a result of using Ada as an implementation language.

3.1.2 Software Preliminary Design Phase

Traditionally, software preliminary design would result in a program structure usually in the form of a structure chart, showing the modules, calling relationships, and interface definitions. Ada impacts this phase because of its more network-like structure, containment capabilities, tasking, and separation of specification and bodies.

Modifications were made in three areas: the activities, and products resulting from the activities associated with software preliminary design, the component structure, and the deliverable documents.

The software preliminary design phase was split into two subphases: software architecture modeling, and computer program component modeling. The major products of software preliminary design were changed from a hierarchy chart to diagrams representing runtime environments, package withing, and task relationships. Compiled package specifications became the main design product.

The original component structure would have resulted in computer programs being decomposed into computer program components, then units, and finally routines. To address Ada, units and routines were both replaced by the Ada program component. This new component was defined as any package, task or subprogram. Ada program components may contain or be composed of other Ada program components. Documentation and modeling products would be organized by Ada program component. Although this organizations alleviates the problem of mapping Ada program units to either a unit or routine, the problems associated with potential redundancy of information within the document are difficult to overcome.

The standard documents, Software System/Subsystem Specification, Program Design Specification, and Program Maintenance Manual, were replaced by a single document, the Software Subsystem Design and Maintenance Specification. This document would have a chapter organization patterned after the component structure.

3.1.2.1 Software Architecture Modeling

The main purpose of software architecture modeling is to identify all computer program components, and the physical interfaces between those components. In most cases, the computer program components were the executable Ada programs, but the definition was expanded to include commercial off-the-shelf software executables, databases, (including the data, schema, and access software), and major libraries of subprograms.

Because of stringent security requirements, the number of computer program components was large, resulting in relatively small Ada programs. A number of computer program components were identified as common to several subsystems.

The products of this phase were data flow diagrams representing the run-time relationships among the executable computer program components, interface descriptions, identification of major databases and libraries. Sections of the Software Subsystem Design and Maintenance Specification corresponding to computer program, and computer program component-level information were completed.

3.1.2.2 Computer Program Component Modeling

Once Ada executables have been identified, the next phase defines the structure of the individual Ada programs. Each computer program component has its own main procedure, multitasking and package structure. The purpose of computer program component modeling is to design these structures. The methodology indicates the need to identify all packages, tasks, and visible subprograms, but allows a great deal of freedom as to the techniques used to help in the identification process. This is important due to the diverse nature of the computer program components. It does specify that the products of this activity include a package withing diagram, for representing withing relationships and compilation dependencies, and a task relationship diagram, for representing the calling relationships among tasks. Both diagrams resemble a structure chart. Ada specifications are written and compiled for all Ada program components except for hidden subprograms.

Sections of the Software Subsystem Design and Maintenance Specification corresponding to computer program components and Ada program components are completed.

3.1.3 Detailed Design Phase

The inputs to this phase would traditionally be callable modules and interface information, but due to the modifications resulting from Ada, there is a certain amount of preliminary design effort that was postponed to this phase. Up to now, only visible subprograms have been identified, and no calling structure has been described. This prompted a modification to the detailed design phase where a subphase called software hierarchy modeling was added to the normal logic specification activity. These two subphases are not necessarily sequential activities, for often it is not until a certain amount of logic specification has been attempted that all hidden subprograms are identified.

3.1.3.1 Software Hierarchy Modeling

The purpose of software hierarchy modeling is to completely design the structure of each package, showing all the subprograms and tasks, and the calling relationships between them. The main products from this activity are structure charts for each package, main subprogram, or stand-alone subprogram, and refinement of specifications, as well as the addition of body stubs for every known Ada program component.

Information is added to the Ada program component sections of the Software Subsystem Design and Maintenance Specifications during this activity.

3.1.3.2 Logic Specification

The purpose of logic specification is to design the algorithms within the Ada program component bodies. Ada is used with certain restriction as a design language for logic specification. The intent being to use Ada to describe as much of the body to enable implementation. All bodies should then be compiled.

Information is added to the Software Subsystem Design and Maintenance Specification for each Ada program component.

3.1.4 Software Implementation Phase

During the software implementation phase, all coding is completed, and each Ada program component is tested at the package level.

The code is debugged, and prepared for integration within each computer program component.

4 Curriculum

The Ada life-cycle training curriculum was designed specifically to provide:

A. Information in a meaningful order, such that whatever knowledge is required to understand a concept has been previously presented.

B. Information based on the life-cycle phase activities as described in the GTE Ada development methodology.

C. Hands-on experience with the techniques required to successfully perform life-cycle activities.

D. Hands-on experience with the GTE Ada Program Support Environment tools required to successfully perform life-cycle activities.

E. Courses tailored for a particular audience.

To satisfy these requirements, the following courses have either been developed or planned:

A. Introduction to Ada

B. Ada Project Management

C. Software Architecture Modeling

D. Ada Design

E. Ada as a Program Design Language

F. Ada Implementation/Test

The characteristics of each of these courses are described in greater detail in the following paragraphs.

4.1 Introduction to Ada

This course covers the entire syntax of the Ada language. Ada is not compared with other languages formally within the course. The decision was made to start with a syntax course because of the need to prepare a number of software engineers for coding tools and prototypes early in the program.

The Introduction to Ada course is presented for four hours every other day over a three week period, and organized to promote understanding by presenting an entire overview of the language and its features on the first day. The next five days are organized so that each successive day presents a more advanced concept relating to types, statements, and program units. Exercises are assigned to emphasize the concepts covered for a particular day. The last three days are concerned with some of the advanced features of Ada, including tasks, exception handling, generics, pragmas, and representation specifications. The three-week format allows students time to attempt the exercises. Students present their exercise solutions to the class to promote discussion and exchange of ideas.

4.2 Ada Project Management

Technical managers require a good understanding of Ada, but the level of detail provided in the Introduction to Ada course may be inappropriate. The intent of the Ada Project Management course is to focus on providing technical managers with Ada knowledge specifically tailored to a managers needs. These needs fall into three areas: the Ada language, Ada development methodology, and project management and supervisory skills. The course is organized into three modules to correspond to these areas.

Module one is a one day overview of the language, focusing on providing technical managers with sufficient knowledge to be able to read and understand Ada specifications. In addition, information regarding why software development using Ada is significantly different than with other languages, and what areas of development are impacted by the use of Ada. These areas include methodology, environment, testing, configuration management, and documentation.

Module two is a two day walkthrough of the Ada software development methodology from software requirements analysis through software integration and test. A detailed view of the methodology, standards, practices, tools, techniques, modeling products, and documents is presented with an emphasis on that information most applicable to the technical manager. Hands-on workshops that focus on understanding development products well enough to evaluate their quality are provided.

Module three is also two days in duration, and covers the administrative aspects of the technical manager role. Topics include planning, schedule and cost control, metrics, organization, supervision, and reporting. There are hands-on workshops relating to planning Ada software projects, reporting status, and supervisory skills.

4.3 Software Architecture Modeling

The Software Architecture Modeling course is eight-hours in duration, and presents information regarding Ada tasking, system services, security, and modeling standards and practices to support the software architecture modeling sub-phase of software preliminary design. The focus of the course is to define the computer program component, and criteria for their identification. In addition, information is presented regarding definition of run-time interfaces between computer program comoponents.

4.4 Ada Design

This three day course is intended to provide the software engineer with an understanding of the activities associated with computer program component modeling subphase through a discussion of Ada design philosophy, software engineering principles, Ada design techiques, and how those techniques map to the Ada development methodology being used. These techniques include object-oriented, function-oriented, data structure-oriented, and process-oriented techniques. Examples of package, subprogram, generic, and task uses are presented. Hands-on, design team workshops are used to provide practice in use of the design techniques. Reviews of the team designs are performed using software engineering principals as evaluation criteria. Design documentation is also discussed.

4.5 Ada as a Program Design Language

This two day course is geared towards providing the necessary skills to perform activities associated with ther subphases of software detailed design. This includes software hierarchy modeling, and logic specification using the Ada language as a program design language. The focus of this course is to provide information regarding modeling techniques for designing the structure of each package in terms of the calling relationships between the subprograms and tasks within a package, and the logic within the bodies of these Ada program components. Hands-on workshops relating to each of these activities are provided.

4.6 Ada Implementation and Test

This two-day course covers the activities relating to the software implementation phase. Topics covered include information necessary to complete implementation of all Ada program components, and the testing of those components. Development of test software, execution of tests, and use of the debugger are also discussed. Hands-on workshops supporting these activities are provided.

In order to enhance understanding, a common workshop example is used for all courses except the Introduction to Ada course. In this way, software development can be followed throughout the life-cycle, thus providing a necessary sense of cohesiveness to the students understanding of how to develop Ada software from beginning to end.

5 Assessment of Training

The intent of this section is to record observations and feedback from the training curriculum received up to December, 1987. At this time, the Introduction to Ada course has so far been presented nine times to a total of 168 software engineers. The Software Architecture Modeling course has been presented five times to a total of 90 software engineers. The Ada Design course has been presented seven times to a total of 105 software engineers. The Ada as a Program Design Language course has been presented three out of a scheduled six times to a total of 48 software engineers. The Ada Project Management course is scheduled to be presented starting in December, 1987 and the Ada Implementation/Test course is scheduled for development starting in January, 1988.

The Introduction to Ada course has proven to be very effective in preparing software engineers with the mechanics of writing Ada code. Completion of the exercises is essential for gaining this understanding. To promote this, certificates of completion are awarded only to those students who complete all exercises. Students are not required to complete the exercises by the end of the course, and they may elect to submit exercises any time after the course is over. The organization of the course has generally been very helpful to most students. And most students concur that the Introduction to Ada course is a necessary prerequisite for the design course.

The Software Architecture Modeling course was effective at presenting necessary information to help experienced software architects make sound architecture decisions based on the tasking capabilities of Ada, the limitations of the operating system, and the restrictions of the security requirements. It also helped to inform them about modeling standards and practices. Unfortunately, only ten percent of the course attendees were experienced enough to benefit adequately from the course.

The course was not intended to provide The Ada Design course is informative, but it has not proven to be an adequate substitute for design experience. Unfortunately the more design experience that a software engineer has in another language, the more difficult, that student seems to have adapting to the Ada design philosophy.

Software engineers with less than two years software development experience appear more receptive, but are often unable to relate to abstract concepts of high-level design as a result of their inexperience. The inadequacies of design techiques and methodologies create confusion that is difficult to avoid when most software engineers are seeking a clear-cut method for making design decisions. This training would be more effective if more instruction time were provided and there were a substantial increase in hands-on workshop time.

The Ada Project Management course should have been the first course completed and delivered. An understanding and acceptance of Ada by management is crucial to the success of a project. A lack of this understanding and acceptance creates an atmosphere of confusion, inverted authority, (where management is not as knowledgeable as the software engineers), and resistance. Managers have the least time, yet require the greatest amount of training and information.

The life-cycle approach itself has generally been well received and effective. The apppropriateness of the information, coupled with the timing of its delivery takes advantage of both the timely need of the student and the limitations of the student to retain information that is not essential to the task at hand. We have been very fortunate in locating skilled trainers with development experience to teach these courses. Without them, the training would not be nearly as effective. We have also attempted to respond to unexpected training needs in the following ways:

A. Deliver briefings on specific topics when needed beyond the information covered by the curriculum.

B. Provide consultation to the software development organization to help resolve problems when they occur.

C. Provide self-paced Ada instruction to relieve the difficulties caused by the inability to provide the syntax training in close proximity to the coding effort.

In conclusion, an effective Ada training program needs to be based on the requirements and constraints of the intended audience. This program is very appropriate for a large project, where strict adherence to a methodology and set of standards is essential to achieve a consistent level of quality. Also, training must be tailored to meet specific needs of the development organization, particularly in the areas of methodology, standards, environment, and management. Finally, Ada language training by itself is an inadequate preparation for serious Ada development. The greatest challenge to Ada training efforts is to help establish a software engineering culture that fosters understanding and cooperation for Ada software development efforts.

6 References

Mellor, Stephen J. and Ward, Paul T. Structured Development for Real Time Systems Vol. 1, Yourdon Press, New York, 1985

GTE Software Engineering Methodology 1987

GTE Software Standards and Practices Manual 1987

About the author: Daniel J. Connolly has performed as Manager, Software Engineering Training at GTE Government Systems in Rockville, Maryland since July, 1985. There he is responsible for designing and implementing the Ada software engineering training curriculum, and the coordination of all other training activities for the software engineering organization. Prior to this, he was a training specialist for General Electric Information Services Company, where he taught software engineering to the software organization. He received a B.A. in mathematics from Hofstra University, and a Masters from SUNY at Stony Brook.

A College Credit Certificate Program in the Ada* Programming Language

Fred L. Bierly

The Pennsylvania State University

Abstract

A college credit Certificate Program in the Ada programming language was established at the Pennsylvania State University through the Office of Continuing Education in the Fall of 1984. The program consists of a three course sequence in Software Design Methods, Introduction to Ada, and Advanced Ada. Students are awarded a certificate of completion upon successfully satisfying the requirements of the three course sequence.

After all of the interest being generated about Ada since its initial announcement there still are not a great many educational institutions providing the opportunity to learn the language and its application. The Software Engineering Institute (SEI) at Carnegie Mellon University has provided *four faculty development workshops to develop curriculum* in software engineering. A part of this effort is to consider Ada as an appropriate language to implement software. However, as of October, 1987, there were only two educational institutions [2] participating in using Ada in the undergraduate curriculum. A serious problem associated with its use is finding, in the 137 validated compilers [1], one that can be used in an educational environment.

Because of the lack of existing curricula, four years ago the Pennsylvania State University, through the urging of the Ada coordinator of a local defense contractor, initiated a college credit certificate program in the Ada programming language. The University Office of Continuing Education contacted me to determine if the Computer Science Department was interested in conducting such a program since our department was already offering a certificate in computer applications through their office. After a number of meetings with John Cupak, the Ada coordinator from industry, Susan Richman, a Professor from our Harrisburg campus, Len Holiday, from our Applied Research Laboratory, and I, an initial two course sequence was developed and offered for persons at the defense contractors site. There were approximately 25 students enrolled in this initial effort.

The two course sequence consisted of a course covering software engineering concepts with beginning Ada followed by a couse on advanced Ada concepts. Each course met twice a week in the early evening hours. Upon successful completion of both courses, each student was given a certificate of completion from Penn State through the Continuing Education office. The Ada program committee, previously mentioned, reviewed this initial effort during the summer of 1984 and decided to offer a three course sequence the following year. Each course would last 10 weeks and meet twice each week in late afternoon or early evening. Also, by the fall of 1984, the Applied Research Laboratory had received an equipment grant with a validated Ada compiler. This equipment was used to offer the Ada portion of the three course sequence. (This year we also have a validated compiler at the Computation Center on one of our mainframes.) The program was also opened to all local industries and received enthusiastic support from them. The program consists of a nine credit, three course sequence in Software Design Methods, Introduction to Ada, and Advanced Ada. It has received an award for new programs in continuing education by the National University Continuing Education Association.

The program is designed for individuals who want to understand the concepts of higher order languages and to learn the Ada language. It is recommended for programmers, software engineers, design consultants, system integration engineers and real-time system architects. Students enrolling in the program must have a baccalaureate degree and some programming experience. Upon satisfactory completion of the three course sequence, each student is awarded a certificate by the University. Enrollment figures for the past three years are noted in figure 1.

Semester	Course	Enrollment	Certificates
Fall 1985	Software Design Intro to Ada Advanced Ada	25 37 11	12
Spring 1986	Advanced Ada	12	6
Fall 1986	Software Design Intro to Ada	21 24	1
Spring 1987	Advanced Ada	21	14
Fall 1987	Software Design Intro to Ada	9 15	

Figure 1.

* Ada is a registered trademark of the United State Government (Ada Joint Program Office.)

of the program objectives by students who are working at local industries during the day but wish to further their education. This typically means providing a more informal and relaxed atmosphere in the classroom and allowing for those persons who have to miss class to take trips for their company an opportunity to make up the class work. There may also be instances of on-the-job deadlines which can interfere with classwork and should be allowed for. That is not to say the students aren't getting a first class education because we feel that they are. We just want to give them an opportunity to participate in the program and make reasonable allowances for them to do so.

In the three credit Software Design Methods course, students are expected to establish specifications describing what a software product is to do; use appropriate tools and techniques for software design to meet software requirements of a system; develop a software product using strategies as determined by current practice; provide for efficient and effective testing of a software product; consider, as part of the implementation, requirements for maintenance of a software product; and establish techniques appropriate for the management of the software life cycle. Although Ada is not used in this course there is a brief discussion of the language as it relates to the design process. Figure 2 is an example of the syllabus used in the Software Design Methods course.

CMPSC 497A Software Design Methods

Description: Application of scientific knowledge and methods in the design and construction of computer software systems with associated documentation.

Textbook: Fairley, Software Engineering Concepts, McGraw-Hill, 1985.

Course Objectives: To prepare students to apply some basic engineering methods to the development of software systems as well as become aware of current trends stated in the literature concerning software engineering.

Outline

1) Introduction to Software Engineering
   a) Definitions
   b) Problems with software
   c) Managerial issues

2) Software Design
   a) Design concepts
   b) Design representations
   c) Design techniques

3) Software Development
   a) Approaches to programming
   b) Programming language features
   c) Software tools
   d) Module development

4) Software Testing
   a) Module test
   b) Program test
   c) System test

5) Software Maintenance
   a) Documentation
   b) Preparing for design
   c) Source-code metrics

6) Software Management
   a) Cost estimation
   b) Managing design
   c) Quality assurance
   d) Managing development
   e) Managing implementation

Figure 2.

The first three credit Ada course, Introduction to Ada, uses a fully validated compiler on a mainframe. Emphasis is on developing well-structured and readable programs that are examples of application from various disciplines. The concepts of modularity, transportability, and separate compilation are stressed. To illustrate the appropriate use of Ada's features to enhance readability and modifiability, students modify their own code and the code of other students to meet changes in the requirements. Upon completion of this first course, students will be able to code "medium-sized" programs using all the features of Ada with the exception of tasking and low-level I/O. Figure 3 is an example of the syllabus used in the Introduction to Ada course.

CMPSC 497B INTRODUCTION TO ADA

Description: Fundamental characteristics of the Ada language including a historical overview of the language's development. Software development methodologies to system design with Ada are presented.

Textbook: Barnes, Programming in Ada, 2nd ed., Addison-Wesley, 1985.

Course Objectives: To expose students to the syntax and semantics of the Ada programming language; to develop proficiency in programming simple applications in Ada stressing hands-on experience through programming assignments.

Outline

1) Basics of Ada programs
  a) Simple types
  b) Introduction to the operating system

2) Overall Ada Structure
  a) Operators and attributes
  b) Type character
  c) Resolution of ambiguities

3) Ada Control Structures
  a) If ... then ...
  b) Loop ...
  c) Case ...
  d) Exit and goto

4) More Control Structures

5) Functions and Procedures
  a) Ada tools
  b) Libmanager
  c) Pretty printer
  d) Interactive debugger

6) Implementation Considerations
  a) Pragmas

7) Introduction to Access Types

8) Introduction to Tasking and Concurrency

Figure 3.

Continuing with the three credit Advanced Ada course, the students emphasize concurrent programming topics and the use of Ada in the software development life cycle. At the end of the course, students should be able to design systems that require concurrency and make design decisions with respect to time and space restrictions on an embedded real-time processor. Figure 4 is an example of the syllabus used in the Advanced Ada course.

CMPSC 497C ADVANCED ADA

Description: Advanced features of Ada including tasking models, buffer tasks, transport tasks, and guard tasks. Representation specification as applied to the underlying machine is also presented.

Textbook: Barnes, Programming in Ada, 2nd ed., Addison-Wesley, 1986.

Course Objectives: To introduce students to the advanced features of the Ada programming language; to develop proficiency in working in a team to program an actual Ada system to include design of such and demonstrations to "users." To bring to the students' attention the practicalities of Ada compiler systems on various machines.

Outline

1) Review of Visibility
  a) Separate compilation
  b) Generic instantiation

2) Encapsulation

3) Tasking for concurrency

4) Tasking constructs
  a) Task initiation
  b) Task scope
  c) Task rendezvous

5) Task Types
  a) Access variables for tasks
  b) Arrays of tasks
  c) Task attributes
  d) Task size control

6) Introduction to Advanced Types
  a) Discriminants and customizing records
  b) Access types
  c) Private types and constraints
  d) Derived types
  e) Numeric types

Figure 4.

There are some problems with offering this sequence in this environment. Doing a three course sequence on a two semester basis creates some scheduling problems. This is especially true for students during the second course which is scheduled over the regular Christmas break. Providing this opportunity for persons in industry also creates some conflict with work assignments. Students may have to take a trip which causes them to miss class or perhaps drop the course or they have a major project deadline to meet at work and can't find the time to complete their studies over a short period of time. But overall, we feel the program has been well received by the majority of the stucents. We also feel this certificate program has been successfully received by industry as meeting the needs of employees involved with defense contracting and those who wish to prepare for the future of higher order languages. The three course sequence also provides the students an opportunity to learn the material required of the courses as well as meet their commitments on the job.

REFERENCES

[1] Ada Information Clearinghouse Newsletter, Vol. V, No.2, September, 1987.

[2] Faculty Development Workshop Report, Software Engineering Institute, Carnegie Mellon University, 1987.

Dr. Fred L. Bierly
220 Whitmore Laboratory
Penn State University
University Park, PA 16802

Fred L. Bierly has taught mathematics and computer science courses since 1964. He joined the Penn State faculty, Department of Computer Science, in 1982 as coordinator of commonwealth campuses and lecturer. He became assistant for undergraduate instruction in 1985. He has also coordinated certificate programs in computer applications and Ada software engineering for the Office of Continuing Education at University Park.

Topics of the Data Analysis Software System Design in Ada*

David S. Galvin

Hughes Aircraft Company, Radar Systems Group

Abstract

This paper describes several design topics of the Data Analysis Software System (DASS) being written in Ada at Hughes Aircraft Company to support F-14 Radar Set flight test and laboratory tasks. The topics covered consist of an Ada Detailed Design Document standard taken from a tailored adaptation of the classical waterfall model for software development; the implementation of multiwindowing by the Executive Tasking Model; and other issues such as task uncoupling, concurrent memory management, and sharable code in a multitasking environment.

1. Introduction

DASS is currently being developed using the DEC Ada compiler version 1.3 on a DEC MicroVAX II that runs the MicroVMS operating system version 4.5, and is a turnkey, menu input, interrupt-driven system with interleaved windowing supported by the User Interface Service (UIS) workstation graphics software. The system accepts keyboard or mouse input, and allows multiple windows of analysis functions to run concurrently via the Ada tasking model. The development of the system is following an adaptation of the classical waterfall model of functional requirements, design, code and test. One-third of the approximately 15 functions are currently in the test stage. The Ada Design Team consists of seven software personnel, including the author of this paper. The completed source code is estimated to be 50,000 Ada-containing lines, not including comments.

DASS, hosted on a DEC VAXstation II/GPX workstation, is designed to process sampled and partially formatted test data from TK50 tape cartridges for interactive analysis, simulation, and statistical functions, thereby allowing the diagnosis of complex problems. High speed data from Radar interfaces and processors is generated from in-flight Radar performance testing, accumulated on analog tape for sampling by a DEC 8600, and then stored on cartridges for DASS consumption. Using several specifically designed languages to produce tabular and plot output formats, DASS provides the data analyst with a means of assessing the quality of the radar flight and laboratory test data acquired during test scenarios. DASS will be used by radar analysts from the U.S. Navy and Hughes Aircraft Company.

2. Ada Detailed Design Document Standard

DASS represents the first major Ada project for the Hughes Weapons System Evaluation Laboratory at Point Mugu, California, and is therefore providing the team with the unusual opportunity to author their own System Development Methodology in the form of an Ada Detailed Design Document standard. Because the detailed design phase for a software module is of such crucial importance in the creation of a software system, and because of the ongoing discussion of Ada as a Program Design Language (PDL) in the Ada community, the results of the detailed design standard are worthy of discussion.

* Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

The issue of a compilable Ada PDL was of significant importance during the original conception of the standard, as it was one of the primary concerns of the team. Another subject that greatly influenced the project's standard was the team's goal to produce detailed design documents that could facilitate the essential communication between team members during the design phase. There was also a desire to write these documents so that they could eventually augment comments in the final source code, thereby resulting in a partially automatic generation of maintenance manuals. By incorporating these ideas with a considerable amount of refinement, the team developed a standard that is now considered to be productive and fairly mature.

The final Ada Detailed Design Document standard calls for a near-compilable Ada PDL that is prefaced by an English non-procedural description. The standard takes a real-world approach by allowing minor modifications to be made to the PDL while the code develops, but aims to forbid the alteration of the non-procedural English description. This procedure, for the most part, preserves the original design, and at the same time produces a document that can be used at a later date in a maintenance environment.

2.1 The Waterfall Adaptation

The classic waterfall method in its original form states that the evolution of a system should proceed from a functional requirement to a detailed design, and then from code to testing. It further states that no reversing of this sequence of events should take place, hence the term "waterfall", implying an obviously irreversible procedure. Much of the discussion concerning system development methodologies has been that the waterfall mechanism is perhaps too overly restrictive, and that a slight relaxing of the procedure would be an appropriate method for developmental guidelines.

The Ada Detailed Design Document standard agrees with this view and additionally recognizes that although desirable, it is not always possible to have 100 percent of the functional level mapped out, particularly in an environment where the authors of the detailed designs are not necessarily the authors of the functional requirements.

There are also occasions where functional requirements that were originally intended to be unrelated become related by some outside factor. If a detailed design stage is in progress and is following one of the above functional requirements, then the detailed design may have to be adjusted to incorporate the additional requirements from the newly related functional requirement document.

For these reasons, a more dynamic [Abb86] detailed design standard was used; specifically, one that permits some degree of alteration as the project progresses. The standard considers minor modifications of an initial design to be part of the design phase, and therefore the documents do not explicitly preserve the design transformation, as is more formally described in [Wil83]. Substantial modifications in the design stage are treated as a re-design, and therefore remain in documented form unless the author of the design considers the modification to be unimportant to the overall understanding of the design.

2.2 The Ada PDL

The decision to use Ada as the base language for the design document standard's PDL was not difficult, because of the large amount of praise the language has received from the community. Ada has been successful when used as a PDL for Ada projects and, equally so, for non-Ada developments*.

The use of a compilable PDL was not as obvious. Compilable PDLs have met with differing opinions, as they provide definite advantages, though not without some documented disadvantages. The deciding argument was that a strictly compilable PDL, while containing advantages through the liberal use of routine stubs, can also lose some flexibility as a result of the lack of natural language English statements [Ber85]. It was decided that requiring a near-compilable PDL would enforce a well-known overall structure, and that comments within the PDL should be permissible to allow the use of psuedo-Ada-English statements in areas where a routine call would otherwise be too restrictive.

It can be argued that allowing comments within the PDL could cause an entire psuedo-code design to pass through the compiler unscathed. Such is an extreme case. The more typical PDL takes advantage of Ada flow control constructs, and employs commented psuedo-code when it is felt that the extra flexibility present in natural language is necessary.

Although it is too early to draw conclusions, it is believed that the format of the PDL can be

* Good results were obtained from an earlier FORTRAN project at the Weapons System Evaluation Laboratory that utilized Ada as the project's PDL.

beneficial during the maintenance stage by "matching" final source code to the PDL in the design document. The code and PDL should synchronize at the flow control statements, with the PDL giving a high level insight into the corresponding blocks of statements in the code.

2.3 Prefacing the Ada PDL with Non-Procedural English Text

A widely held belief regarding software development is that the communication between designers during the design stage is very possibly the most important part of the process. The essential exchange of ideas very often occurs through the composition and review of detailed design documentation.

This universally accepted regard for good communication of ideas established a need for the Detailed Design Document standard to include English text that describes, in a fairly formal non-procedural manner, the mechanics of the PDL. This description, if complete, should be capable of allowing an implementation to follow directly from the text.

A non-precise definition of non-procedural text, taken loosely from database principles [Ris86], is to describe an operation by means that do not state the explicit ordered steps needed to achieve the operation. The intent of a non-procedural introduction for each module in the design document is to direct the focus on the routine's desired behavior without making detailed and structured statements about how the results are to be achieved. As an example, the typical statement about the initialization of an array need not mention the order of the individual cell initializations, provided the described algorithm does not depend on this particular order. Although an array is a trivial example, most complicated algorithms can be described non-procedurally, and require a significant amount of well-spent effort and thought. Much of the English language is inherently non-procedural, however, the added restriction that no procedural text must be present in the description helps to ensure that the introductory text does not start to describe itself in procedural terms, thereby repeating the approach to the solution that can be found in the PDL.

This format consisting of a procedural Ada PDL with a non-procedural introduction, has a striking resemblance to the concept of the Specification and corresponding Body of an Ada unit. The similarity is not surprising: the specification part of an Ada unit is considered its "contract" with its surroundings, and its body is capable of expressing one of several possible implementations satisfying the requirements of the specification. This is not at all a bad way to view a detailed design. This scheme not only provides the much needed interface between team members, but also maintains local integrity within the individual components of the designer's module when a similar approach is applied.

3. Executive Tasking Model and Concurrency Topics

The requirement that DASS should be a multiwindowing, multitasking system resulted in the extensive use of the Ada tasking model. Because DASS is required to have a reasonable interactive response along with a concurrent high performance of intensive test data analysis functions, the implementation of this unusual combination is especially noteworthy.

DASS consists of functions that use pairs of windows for interactive control and for data analysis processing and subsequent output. The functions are managed by a single master function known as the Executive. It is the Executive's responsibility to create, monitor, suspend, and abort all of the system's functions. The method of communication, as alluded to earlier, is via the Ada tasking rendezvous mechanism. The Executive is implemented as a task that starts, and that is subsequently interrupted by, analysis functions also implemented as tasks. As new functions are requested or when a condition changes, DASS functions are therefore permitted to notify the Executive of their status through the rendezvous.

Additional communication must exist between the Executive and the analysis functions due to the fact that a completely occluded window is a common occurrence, as DASS can support many layers of windows. One of the possible rendezvous with an analysis function must be a request for the window to make itself completely visible. This communication is necessary to provide the Executive with control over the presence of an analysis window, and possibly some control over other sub-windows created by the analysis function.

Analogous to the considerations of the system's communication for its analysis functions is a set of relatively low-level Ada packages that manage interactive window input. These are generic packages that act as interfaces between keyboard asynchronous system traps and interactive windows, and cause routines to be notified (via the rendezvous mechanism) when input for a particular window is present. The overhead of using tasking facilities for interactive character processing was not found to be a problem for the windows on an individual basis, and did not significantly burden the rest of the system.

The Executive Tasking Model proved to be ideal for this application and is used as a template by some functions that need to control certain sub-functions. However, the incorporation of tasking adds a significant amount of complexity to the overall system. Several relevant topics of interest follow.

3.1 Task Coupling

In order for the DASS analysis functions and the rest of the necessary system tasks to operate cooperatively in a concurrent environment, a high degree of asynchronous execution must exist for data retrieval and other related operations. It is obviously unacceptable for one function to be forced to wait for its requested data until another function has received all of its desired data. Furthermore, it is also usually undesirable for an individual function to have to wait, for even short periods of time, for its own possibly synchronous data collection. Ideally, some portion of the anticipated data should be (virtually) immediately available per request.

This type of desired behavior necessarily implies a loose amount of coupling between tasks in order to maximize the amount of asynchronous operation. Uncoupling can be achieved through the use intermediary tasks [Nie86] to provide the appropriate minimal amoun· of synchronous operation.

An appropriate example of task uncoupling is the use of a task to act as a buffer between the data gathering tasks and the analysis functions. This can be viewed as the traditional producer-consumer situation. An intermediate task is employed to uncouple both the consumer and producer and to provide the necessary mutual exclusion to alleviate the contention problems that can arise with asynchronous producer-consumer operations. Because the Ada rendezvous is the primary means of task synchronization and communication, it is the ideal mechanism to ensure only synchronous operations from both the producer and consumer. It also provides the sufficient isolation (buffering) to cause a low degree of task coupling, thus reducing the system's overall synchronous execution.

There is added overhead with this scheme [Bur87]. However, it is clearly acceptable when one considers that the concurrent buffering and storing of data allows functions to simply post a rendezvous to receive data, as opposed to waiting for a sequential procedure to process a large amount of data per each call.

3.2 Concurrent Memory Management

The desired high degree of parallel operation in DASS involves some aspects of memory management. The implementation of the system relies on a considerable amount of dynamic memory allocation and subsequent de-allocation. The VAXAda compiler does not implement automatic system "garbage collection"; the Ada Language Reference Manual [Ref83] states that an implementation need not perform its own reclamation of objects that are no longer referenced. However, the generic procedure **unchecked_deallocation** is available, and is therefore used to occasionally return large amounts of unwanted memory to the system for probable re-use.

(As an aside, some information that pertains to memory management is worth introducing, although the remainder is not within the scope of this paper. Not all compiler vendors implement **unchecked_deallocation;** those that do, exhibit significant differences. One might at first assume that memory is allocated from and returned to some kind of central pool of storage space. Although this is the case for some compilers, others, including VAXAda, have incorporated a "typed-based" storage scheme in which memory is only de-allocated back to storage pools belonging to a specific type. In other words, memory once allocated to an object of type **T**, when subsequently de-allocated, will only be made available to further allocations for objects of type **T**. This has some interesting properties that will not be discussed further*.)

Because the number of dynamically allocated blocks of memory can be significant, the amount of time spent de-allocating the space should be considered, particularly when the blocks are arranged in the common linked- list format. The process of de-allocating memory is traditionally a sequential activity, and is therefore a prime candidate for the de-allocation of memory in a concurrent fashion, which would once again reduce the amount of coupling in the system.

The concurrent solution is the use of a task that accepts a list of blocks of memory and performs de-allocation on the blocks without

* Currently the author and others are investigating the possibility and benefits of implementing the central-pool approach of decallocation that avoids the limitation of type-based de-allocation through the use of a generic memory management package.

further communication with the routine that initiated the request. An entry is provided so that additional amounts of memory can be appended to the existing list of blocks, in the event that more unwanted memory arrives before all of the original blocks are de-allocated. The process of initiating the de- allocation of blocks of memory is a single rendezvous to transfer the beginning address of the unwanted memory to the task.

Obviously some discretion must be used before this utility task is used. The time required for a rendezvous is a non-zero amount. It is probable that for small amounts of memory, the added overhead of the rendezvous will make regular sequential de-allocation the preferable choice. However, for larger quantities of memory, the task de-allocation method has definite benefits.

An enhancement that can be made to the above algorithm is the inclusion of the **priority** pragma within the de-allocating task to cause the task to execute with a lower priority. The motivation depends on the assumption that in many cases memory de-allocation does not need immediate attention and can be accomplished at a slower rate, thereby giving additional central processing unit time to more deserving tasks.

The use of different priorities is currently under consideration. There exists the possibility of "priority inversion" [Sha87], in which tasks of differing priorities can have their priorities effectively switched due to the current language implementation of the rendezvous. A recommendation has been made to propose a "priority inheritance" scheme for the upcoming 9X revision of the Ada language [Cor87]. As mentioned earlier, the possible consolidation of memory pools is under current investigation. This consolidation might discourage a reduced rate scheme for memory de-allocation, because reclaimable space from a central shared area is likely to be in more urgent demand.

Taking the above issues under consideration, the **priority** enhancement appears premature at this time. However, the use of the original de-allocating task can still be utilized to add some degree of parallelism to the frequent operation of memory de-allocation, and provide lower task coupling throughout the system.

3.3 Sharable Code

The Ada reference manual states that procedures and functions are to be reentrant. It follows that no side effects can occur when such a routine is invoked by multiple processes simultaneously, as each invocation operates on an individual copy. However, packages do not necessarily have the protection of reentrancy; routines that are declared in the specification and/or body are individually reentrant, although their usage may not be sharable when used in the context of a package that employs persistent data.

It is not uncommon for the implementation of a package to rely on data that retains its value between successive calls to one or more of its routines. (The idea of persistent data in subroutines was first explicitly addressed by Algol 60 [Ran64], [Coh85], through a type of variable known as **own**). Persistent data is typically implemented as object declarations residing in the package body, thereby allowing the logical scope of the identifiers to exist between calls to the package. This method for packages can be appropriate only when it is used in a non-multitasking situation. Asynchronous access to an unsuspecting package in a tasking environment can produce erroneous results.

Many solutions in the literature include discussions of sharable packages that provide multiple access protection for the same information. An example are processes executing in parallel that want timely access to a single queue or stack, etc. The incorporation of a task inside a package can facilitate the necessary mutual exclusion of the routines operating on a shared resource by limiting the access to the routines through a rendezvous. A package of this kind is known as a "monitor" [Coh85], and contains a task that places each of the package's routines inside an **accept** block, so that only synchronous access to a package routine is possible.

Little is said about utility packages that do not want to use data structures common to other calling routines, and that need to provide a service that is completely independent and isolated from all other areas of the program. The need for packages of this latter variety emerged in DASS because of the concurrency present through the use of the Executive Tasking Model and from the use of utility data gathering routines.

The solution used here is known as a "type manager" [Int]. A type manager is a package that declares no storage space for objects in its specification or in its body, and instead only has declarations for types, subtypes, routines, and objects declared as **constant**. The package specification also includes the type declaration of a single record structure that is capable of holding all the necessary persistent data inside fields of the record. It is with this structure that all needed persistent data is transmitted into and out of the package routines by including the type name of the record as the (typically) last parameter in a routine's parameter list. Note that this is still only a type declaration in the package specification; the object of this type is declared in all routines that wish to use this package, i.e., any unit that has **withed** the given package. Thus, the storage for the object resides with the user code, safe from all other concurrent users of the package, because the object is locally declared.

Other less interesting solutions include a generic approach that forces a package to become generic so that its use requires an instantiation, thus providing individual copies to processes that require its services. Another solution gives the package in question the additional responsibility for receiving some kind of unique identification from calling routines. The package can then use this information to protect processes from each other by creating an ID-keyed environment for each ID received.

Although other solutions do exist, the type manager appears to be by far the more elegant and convenient solution for concurrently accessed packages when compared to solutions that require additional logic, or implementations that necessitate the extra memory needed for generic instantiations.

4. Conclusion

The issues covered in this paper represent some of the more pertinent topics pertaining to the design and implementation of DASS. The Ada Detailed Design Document standard presented effectively serves both the design and the maintenance stages. It provides the designers with the necessary tools for a suitable method of design by making appropriate use of a compilable PDL format and a fairly formal program module description. The standard also addresses the need of useful documentation for the maintenance stage by generating these documents from the evolution of the original design.

The presented Executive Tasking Model satisfied its multitasking requirements, and encouraged the generally desirable low coupling of modules through the use of intermediate tasks. Memory management concerns were involved because of a need to maintain a sufficient degree of concurrency. Also discussed is a useful method of memory de-allocation, as well as an introduction to a possible new implementation of memory management in the future. The use of packages in a multitasking environment is discussed, and a successful solution is provided.

Acknowledgments

The Ada Design Team (D. Banks, A. Borisch, J. Dietch, D. Galvin, C. McBride, G. Slivinske, D. Thieme).

References

[Abb86] Abbott, R. "Approaches to Software Development", program notes from ACM SIGSoft Seminar, October 1986.

[Abb83] Abbott, R. "Program Design by Informal English Descriptions", Communications of the ACM, Volume 26, Number 11, November 1983.

[Bak85] Baker, T., Riccardi, G. "Ada Tasking: From Semantics to Efficient Implementation", IEEE Software, Volume 2, Number 2, March 1985.

[Bar84] Barnes, J. "Programming in Ada", Addison-Wesley, 1984.

[Ber85] Berry, D. "On the Requirements for and the use of a Program Design Language: Parameterization, Abstract Data Typing, Strong Typing", Computer Science Department, University of California, Los Angeles, May 1985.

[Boo83] Booch, G. "Software Engineering with Ada", Benjamin/Cummings, 1983.

[Bur87] Burger, T., Neilson K., "An Assessment of the Overhead Associated with Tasking Facilities and Task Paradigms in Ada", ACM SIGAda Ada Letters, Volume 7 Number 1, January 1987.

[Coh85] Cohen, N. "Tasks as Abstraction Mechanisms", ACM SIGAda Ada Letters, Volume V Number 3, November 1985.

[Cor87] Cornhill, D. "Tasking", International Workshop on Real-Time Ada Issues, session summary, Moretonhampstead, Devon, UK, May 1987.

[Gor87] Gordon, R. L. "Window Management, Graphics and Operating Systems", ACM SIGOPS Operating Systems Review, Volume 21 Number 3, July 1987.

[Hab83] Habermann, A., Dewayne, E., "Ada for Experienced Programmers", Addison-Wesley, 1983.

[Loc87] Locke, C., D. Vogel, "Problems in Ada Runtime Task Scheduling", International Workshop on Real-Time Ada Issues, Moretonhampstead, Devon, UK, May 1987.

[Nie86] Nielsen, K. "Task Coupling and Cohesion in Ada", ACM SIGAda Ada Letters, Volume VI Number 4, July 1986.

[Ran64] Randell, B., Russell, L. "Algol 60 Implementation", Academic Press, 1964.

[Ref83] "Reference Manual for the Ada Programming Language", ANSI/Military standard MIL-STD-1815A, US DoD, January 1983.

[Ris88] Rishe, N. "Logical Database Design", to be published in 1988, Computer Science Department, University of California, Santa Barbara.

[Sha87] Cornhill, D., Sha, L. "Priority Inversion in Ada", ACM SIGAda Ada Letters, Volume VII Number 7, November 1987.

[Wil83] Wile, D. "Program Developments: Formal Explanations of Implementations", Communications of the ACM, Volume 26 Number 11, November 1983.

[Yue85] Yuen, C. K. "On Programs, Tasks, and Processes", ACM SIGOPS Operating Systems Review, Volume 19 Number 3, July 1985.

[Int] Personal Communication: Dr. B. Bardin; clarification of Intel's origination of the notion of "type manager" through early work with the 432 Processor.

Mr. Galvin holds a B.S. in Mathematics and Computer Science from the University of California at Los Angeles, and has been a member of the Ada Design Team at Hughes Aircraft Company since 1985. He has completed computer science courses at the University of California at Santa Barbara, and is currently pursuing a M.S. at the University of Southern California in Computer Engineering in the area of system performance evaluation and distributed processing.

Current mailing address:

Hughes Aircraft Company
Radar Systems Group F14 Division
Weapons System Evaluation Laboratory
Pacific Missile Test Center
P.O. Box 42202, Point Mugu, CA 93042

Standard Ada* and Organizational Dialects

Cynthia P. Baehr

GTE Government Systems Corporation, WIS Division

ABSTRACT

Members of distinct organizations tend to develop different views of how best to use Ada. Software engineering groups develop their own coding standards and design methods, which result in distinct dialects of Ada.

This paper discusses various ways in which Ada code may differ, showing examples of a system that was designed and coded by engineers using three different methods of design and implementation. These examples are compared to show how the different coding standards and design methods can result in the development of different Ada dialects. Documentation is proposed to include these guidelines and methods, so that an engineer from a different organization may better understand the dialect used.

I. INTRODUCTION

Ada syntax is very strictly standardized, and valid Ada compiles on any validated compiler. This allows Ada to be reusable and portable, in order to reduce duplication and software costs.

Nevertheless different organizations write distinctive Ada. Different constructs are favoured, pragmas are used differently, and the code is derived from different architectural approaches. As a result, it becomes difficult for members of other organizations to understand off-the-shelf Ada code and for itinerant programmers to get up to speed on Ada systems, even after several years of Ada experience.

II. ORGANIZATIONAL DIALECTS

A. Reasons for Dialectal Differences

The driving forces towards development of dialects include the design methods used by an organization, the coding standards and guidelines provided to the programmers, and the factors of portability and efficiency desired by the organization.

1. Design Methods

Different organizations use different methods when designing software systems. These may include structural methods, object-oriented design, and functional decomposition among others. Each method will produce a different design which will be packaged and implemented differently. A top-down structural approach will tend to result in tighter, less reusable code. An object-oriented approach will tend to result in reusable, though somewhat verbose, code. Functional decomposition is a well known approach and will offer some reusability.

2. Coding Standards and Guidelines

The guidelines of an organization reflect its philosophy on the use of Ada with respect to concepts such as reusability, efficiency, or portability. They are also influenced by the customer's requirements, the size of the computer system being used, and the software system being developed. These guidelines may be thoroughly spelled out in manuals or may existy by cultural feedback. They are enforced by supervisors and by peer review, and the stricter the enforcement, the more homogeneous the dialect.

3. Emphasis on Portability

Most Ada is portable due to the standardization of the language, but the availability of various pragmas, unchecked conversion, and unchecked deallocation can change from compiler to compiler. The importance of portability to an organization (or its customer in a given instance) depends upon the future uses of an entire system or the reusability of specific library units. The portability of a unit depends upon how close the functionality is to the hardware.

*Ada is a registered trademark of the US Government (Ada Joint Program Office).

4. Efficiency

Various needs for efficiency will change a design. To save space, libraries of utilities and generics may be used by different systems. If this is not a problem, the code may be designed to be verbose and therefore more readable, which may improve maintainability in the future, but may take longer to implement. A need for runtime speed will also affect the method used, as the designer may try to cut down on the number of subroutine calls.

5. Local Character

Beyond the printed guidelines, a local character develops due to the choice of text for learning the Ada language, the functionality of the available compiler, or the desire within a group to keep the code as portable and as reusable as posible. As a group of engineers develops various systems in Ada, they help one another by reviewing the design and code. By doing so, they learn from one another, and start to use the same constructs within their code. So a group will start with a common philosophy, members will learn common constructs and usages from one another, and slowly a dialect will develop.

B. Use of Constructs

The richness of features in Ada allow the same algorithm to be implemented using a wide variety of different structures. So, if a group focuses on the ability to hide information by using private types, their specifications may look quite different from those written by a group which focuses on generics. Once a system is packaged, the engineer must decide which packages need to be 'with'ed, and whether or not to include 'use' statements. Should the types be private? Could a particular package be used as a generic? What is the preferred frequency and manner of using the various primitive Ada constructs (e.g., goto, named exceptions, end labels, subtypes)? Each of these and similar questions could be answered differently depending upon the guidelines and coding standards set up by the group.

1. Conventions

Among the aspects covered in a set of guidelines, naming conventions and pretty printing are likely to be prominent. Naming conventions change readability. Some organizational guidelines give specific rules of capitalization and abbreviation while others leave such things up to the implementer. Pretty printers may be used to unify the readability of all code within a organization, but these vary to the extent that identical code may look quite different depending upon the printer used.

2. Packages

While packaging in Ada allows for orderly grouping of system code, the standard for Ada offers no requirements as to the method of grouping used. Depending upon the design method used and the recommended guidelines, the packaging of a system may vary widely. One group may choose to use one package for all global types, data structures and variables, while another may require that each package contains no more than one type. The nesting of packages is allowed in Ada, but some groups feel that they cause confusion, so may restrict their use.

3. Generics

Different outlooks on the use of generics can affect the way a system is designed. One may push towards a universal utility for generic packages, while others choose to use them in a more localized setting by incorporating generic subprograms within non-generic packages.

4. Tasking

The various methods of setting up the tasking within a system can change the way it is structured. For example, one group may choose to set up several different tasks, while another uses several entries into a single task. Tasks may be started within the main procedure of a program, or started at elaboration time. Other practices which will affect the way a system is packaged, and how it runs, include the tightness of the rendezvous code, the use of semaphoring, and how hardware interupts are serviced.

5. Private Types

An organization's philosophy of Ada influences the frequency of use of private types. The extensive use of private types allows the protection of data integrity, but it restricts the way the data is accessed, thereby changing the code produced. Selective use may loosen the protection of the data, but may increase modifiability.

6. Pragmas

Different pragmas are available for different compilers, and therefore their use will affect the code. Some groups may choose to use pragma in_line for added speed. If a group has an extensive library of code written in other languages, they will use pragma interface extensively.

7. Exception Handlers

Some groups use exception handlers only for serious errors, while some insist that all exceptions be handled or reraised. They have even been used as 'if' statements and as a means of exiting a loop. One group will raise an exception in a situation where another group would pass a flag. Other differences will arise from choices of named exceptions and the level at which exceptions are handled.

8. Optional Constructs

Certain constructs in Ada can be used explicitly or implicitly. These include such things as 'in', 'other', end labels, dot notation (even with use clauses), and renaming. The choice on use of these constructs differs from organization to organization.

III. EXAMPLES

Consider three organizations tackling the same problem, in this case a system to create a family tree. Each group used the same input file and produced the same output, but each used its own design method, coding standards, and guidelines. The input is a file of records with four data elements each (Appendix I.A.). The output was specified to be an indented tabulation of descendants (Appendix I.B.). The specifications for each of the three systems may be found in Appendix II. The major differences among the implementations derive from the design approaches used. The coding standards and guidelines reflect the design approach as well as the rest of the organizations' Ada cultures.

A. Structural Approach

Example 1 was written using a structural approach and top down design of the necessary functionality. The customer wanted software to produce family trees and was not interested in other applications. As a result the design consists of one package exporting a record type which contains all of the information needed to implement the system, and two procedures to perform the necessary actions. Key features of the organization's coding guidelines are listed in Table 1.

Table 1: Coding Guideline Features for Structural Approach Organization

1. Packages may be nested. Procedure and function nesting should be avoided.

2. Each variable, block, subprogram, etc., should serve a single, simple purpose.

3. Always use end labels on packages, tasks, and subprograms.

4. Limit the number of arguments to a subprogram. Four arguments is about the maximum most people can comprehend.

5. Follow the indenting rules in the Ada Language Reference Manual, with 2 as the unit of indentation.

6. The first letter of each word in a name should be capitalized (i.e., FamilyTree1).

7. Avoid abbreviations. Document and be consistent for the few words abbreviated.

8. Use 'renames' to simplify long dot-selected names.

The guidelines explain the initial method used and describes the packaging of the system. The narrow focus on the software to produce family trees led the engineer to use a simple top-down approach. Since the system was not expected to be used for any other purpose, this was the most efficient method to use so to save time and money. Example 1 would be portable, but would not offer any reusability. The code is written to produce a specific type of family tree.

B. Object-Oriented Design

Example 2 was written using an object-oriented approach. The customer wanted software to produce a family tree, but was also interested in building up a library of software that could be used for other applications using personal information. The designers determined that three different objects were necessary, and the packages reflect these objects: information, family member, and family tree. Each package exports a type corresponding to one of the objects and subprograms which allow manipulation of the objects.

Table 2: Coding Guideline Features for Object-Oriented Design Organization

1. There shall be at most one visible type defined per package.

2. All visible types shall be private, with the exception of enumerated types.

3. No use clauses shall appear in any code.

4. There shall be no nested packages.

5. Named parameter associations shall be used for all subprogram calls.

6. User-defined Ada identifiers shall appear in mixed case, with the first letter of each word capitalized. Identifiers formed with more than one word shall be connected by underscores.

7. All identifiers which are meant to be valid English words should be spelled correctly.

Because an object-oriented method was used, the program is separated into several packages, and it offers some reusablity. There are now three packages which can be used in other systems. For instance, package Information could serve as a template for use in other applications which need to store the information associated with a person, such as to keep information for a doctor's records or a school system which wanted to be able to reach a child's parents. However, because of the number of lines of code needed to implement three packages, this approach is more costly in storage space.

C. Functional Decomposition

Example 3 was designed with a top-down functional decomposition method. The design consists of a generic package from the organization's library, with an instantiation of that package, and two new packages. One of these contains a record for the person's data and the necessary functionality as described in the requirements. The second package contains the subprograms necessary to perform the operations of the system. This was designed as a prototype, without all of the requirements for the final applications.

Table 3: Coding Guidelines for Functional Decomposition Organization

1. Minimize and localize dependencies, especially in specifications.

2. Ensure data integrity through use of record private types and routines to return the values of each field of the record.

3. In naming a generic package, describe the limitations and characteristics of the implementation in the name (e.g., "Single_Unbounded_List" instead of "List").

4. Use standardized abbreviations for package renames for clarity. Use the same abbreviations everywhere within the system.

5. Limit use clauses to one or two per package. Apply use clauses only on packages that should be globally understandable (e.g., Text_IO), not to special packages.

6. Give use clauses for expression evaluation on the smallest possible scope, through use of declare-end blocks.

The designer was told that the generic package Single_Unbounded_List was available. Since all of the requirements for the final applications were not yet ready, the system was liable to significantly change. For this reason, it was not designed with reentrant code (see Make_Tree3 in Appendix IIC), but rather with very few dependencies among the various specifications. Example 3, with its generic package, is oriented towards reusability.

D. Stylistic Differences

Figure 1 shows samples of code from the three examples.

The code in figures 1.a and 1.b was written by the same engineer, using the guidelines for the structural approach organization and the object-oriented organization respectively. Structurally, it is virtually identical, but the different guidelines for each have changed the way it looks. The code in 1.c was written by another engineer using the guidelines for the functional decomposition organization. The implementation and the style have changed.

```
with Text_IO; use Text_IO;
...
procedure Search(Parent : FamilyMemberPtr)
  is
  -- this procedure searches the member
  -- list for the descendants of the
  -- given parent.

  Member : FamilyMemberPtr := FirstMember;

begin
  -- indent the descendants
  PrintCol := PrintCol + 2;
  -- check all the members to find the
  -- children of the given parent
  while Member /= null loop

    if Member.Mother = Parent
    or Member.Father = Parent  then

      if Member.Searched then
        -- if we have found this member
        --before, print the information
        Print(Member);
      else
        -- print the information,
        -- and search for the children
        Print(Member);
        Member.Searched := true;
        Search(Member);
      end if;
    end if;

    Member := Member.Next;
  end loop;

  PrintCol := PrintCol - 2;
end Search;
```

a. With use clauses and implied parameter associations.

```
with Text_IO;
...
procedure Search(Parent : Member_List) is
-- searches for the descendants of
-- the parent

     Member : Member_List
       := The_Family_Tree.First_Member;

  begin

    Print_Column :=
      Text_IO."+"(Print_Column,
                  Indent_Count);

    while Member /= null loop
      if Member.Mother = Parent or
         Member.Father = Parent then

         Member.Column := Print_Column;
         if Member.Searched then
           Print (Member => Member);
         else
           Member.Searched := true;
           Print (Member => Member);
           Search(Parent => Member);
         end if;

      end if;

      Member := Member.Next;

    end loop;
    Print_Column :=
      Text_IO."-"(Print_Column,
                  Indent_Count);
   end Search;
```

b. With extended dot names and explicit parameter associations.

```
...
package pd  renames person_data;
package pel renames person_element_list;
...
procedure get_descendants (person : pd.person_element;
                           indent : integer) is
  child_element : pd.person_element;
  local_handle  : pel.handle;
begin
  local_handle := pel.next(beginning_of_list);
  while not pel.at_end_of_list(local_handle) loop
    child_element := pel.get_element(local_handle);
    if        (pd.has_parent(child_element))
    and then (pd.get_mother(child_element) = pd.get_name(person)
    or else   pd.get_father(child_element) = pd.get_name(person))
    then
      for i in 1..indent loop
        put(" ");
      end loop;
      put_line (pd.get_name(child_element));
      for i in 1..indent loop
        put(" ");
      end loop;
      put (" b. ");
      put_line(pd.get_dob(child_element));
      get_descendants(child_element, indent+4);
      local_handle := pel.next(local_handle);
    else
      local_handle := pel.next(local_handle);
    end if;
  end loop;
end get_descendants;
```

c. With different implementation of similar algorithm.

Figure 1. Stylistic Differences in Code.

IV. DOCUMENTATION AS A SOLUTION

As more software is written in Ada, one would like to be able to port and reuse existing Ada code to help reduce the time and resources necessary to create new software systems. Because of the different methods being used by various groups when writing Ada code, this task is becoming more difficult. Each group is producing valid Ada code, but another group with a different focus may not necessarily understand it.

The solution to this problem is not to standardize on a single design methodology or single set of guidelines. Any standardization beyond the language standard may reduce Ada's versatility. Even within a given organization's set of guidelines there is room for differences in style, and that should be maintained to allow for each engineer's individual creativity. The solution to facilitating reuse while maintaining the flexibility in the use of Ada appears to lie in strong documentation. Documents that describe the organization's philosophical approach to Ada, to design methodology, and to the overall design goals of a particular system can profoundly influence the engineer's ability to assimilate into a new project environment.

Even the small amount of documentation shown here can be helpful in understanding these designs. For a larger system, each subsystem could have its own document to lead the reader along in his understanding. The documents should also mention the system used for development and the target system, if applicable, either of which may have added constraints to the way that the language was used. Any information which could aid the user in understanding the reasoning behind the design and implementation can make the system more understandable and will, therefore, help the user to determine whether this is the system he wants to use in his application.

IV. SUMMARY

Even an experienced Ada user, when he changes organizations, may have trouble understanding the software previously developed on his new project. Conversely, users trying to incorporate software written in another organization may have similar difficulties. Ada code written by one group may differ considerably from code with the same functionality written by a different group. Each software house has different design methodologies, different coding standards, and even different ways of utilizing Ada constructs. Syntactically correct Ada code compiles on all Ada compilers, but if the user is unfamiliar with the original developer's philosophies and coding guidelines, he may not be able to understand the specifications or the code, and he may choose not to reuse it in a new system.

One way to make systems and compilation units more understandable is to provide documentation to allow the user more insight into the original designer's approach. It has always been recognized that the documentation of code through comments is important for software readability, but with knowledge of the original requirements and the design method used, the software is more likely to be appreciated and used.

V. ACKNOWLEDGEMENTS

The author is grateful to the CASE Department at the GTE WIS Division for their support on this project, especially to Kathy Kroenert for her encouragement. The author is also grateful to Rachel Silber who helped develop the different solutions to the example system.

VI. BIOGRAPHY

Cynthia P. Baehr is a Member of the Technical Staff at GTE WIS Division in Billerica, MA, where she is working with the Computer Aided Software Engineering Department. She has been working a wide range of Ada projects for the past five years. Ms. Baehr received her B.A. degree in Mathematics from Wellesley College.

GTE, WIS Division
1 Federal Street
Billerica, MA 01821

Appendix I

A. Input File

The input file consists of information about the family members. Each member has four pieces of information: his name, his date of birth, his mother's name and his father's name. If the information is not known, a questıon mark is used.

```
Megan
03-17-85
Cindi
Tom
Cindi
09-19-57
Jeanne
Charles
Charles
04-14-25
?
?
Jeanne
03-28-27
?
?
Martha
10-29-54
Jeanne
Charles
Karen
12-08-55
Jeanne
Charles
Erica
05-20-81
Karen
Dana
frank
06-14-82
Martha
Jerry
Edward
06-18-84
Martha
Jerry
Andy
11-06-57
Jeanne
Charles
Joanne
05-06-84
Karen
Dana
```

B. Output File

The output was was the asme for each system. This output shows each family member and his decendents. Formatting for the output is controlled by the bodies of the Family_Tree packages.

```
Charles
 b. 04-14-25
  Cindi
   b. 09-19-57
    Megan
     b. 03-17-85
  Martha
   b. 10-29-54
    Frank
     b. 06-14-82
    Edward
     b. 06-18-84
  Karen
   b. 12-08-55
    Erica
     b. 05-20-81
    Joanne
     b. 05-06-84
  Andy
   b. 11-06-57
Jeanne
 b. 03-28-27
  Cindi
   b. 09-19-57
  Martha
   b. 10-29-54
  Karen
   b. 12-08-55
  Andy
   b. 11-06-57
```

Appendix II. Package Specifications

A. Specification for Example 1

```
package FamilyTree1 is

  subtype Date is String (1..8);
  subtype NameString is String (1..25);

  NullName : NameString := (others => ' ');
  NullDate : Date := (others => ' ');

  type FamilyMemberType;
  type FamilyMemberPtr is access FamilyMemberType;
  type FamilyMemberType is
    -- this record holds the information
    -- for each family member.
    record
      Name : NameString := NullName;
      DateOfBirth : Date := NullDate;
      MotherName : NameString := NullName;
      FatherName : NameString := NullName;
      Mother : FamilyMemberPtr := null;
      Father : FamilyMemberPtr := null;
      Searched : Boolean := FALSE;
      Last   : FamilyMemberPtr := null;
      Next   : FamilyMemberPtr := null;
    end record;

  procedure ReadFamilyInformationFile;
    -- this procedure will ask the user for an input file name, and
    -- read the family information from the given input file.

  procedure ProcessAndReport;
    -- this procedure will sort the information, ask
    -- for an output file name, and print the Family
    -- tree in the given output file.

end FamilyTree1;

-------------------------------------------------------------------

with FamilyTree1;
procedure MakeTree1 is

-- This is the driver procedure for the Family Tree version 1.

  begin
    -- read the information from the input file
    FamilyTree1.ReadFamilyInformationFile;

    -- process the information and produce the report
    FamilyTree1.ProcessAndReport;

  end MakeTree1;

-------------------------------------------------------------------
```

B. Specification for Example 2

```
package Information is

  type Information_Type is private;

  procedure Open_Information_File;
    -- Gets the name of the Input file,
    -- and opens the given file.
  function End_Of_File return Boolean;
    -- Returns true if the input
    -- file is at the end of file.
  function Get return Information_Type;
    -- Reads input from the file and returns the
    -- information in an Information Type

  function Name (The_Information : in Information_Type)
     return String;
    -- Returns the name associated with the Information.

  function Mothers_Name (The_Information : in Information_Type)
     return String;
    -- Returns the Mother's name associated with the Information.

  function Fathers_Name (The_Information : in Information_Type)
     return String;
    -- Returns the father's name associated with the Information.

  function Date_Of_Birth (The_Information : in Information_Type)
     return String;
    -- Returns the date of birth associated with the Information.

  function Null_Date return String;
    -- Returns a null date.

private

  Null_Name : String (1..25) := (others => ' ');
  Null_Date_String : String (1..8) := (others => ' ');

  type Information_Type is
    record
      Name : String (1..25) := Null_Name;
      Date_Of_Birth : String (1..8) := Null_Date_String;
      Mothers_Name  : String (1..25) := Null_Name;
      Fathers_Name  : String (1..25) := Null_Name;
    end record;

end Information;

-------------------------------------------------------------------

with Information;
package Family_Member is

  type Family_Member_Type is private;

  function Null_Member return Family_Member_Type;
  -- Returns a null member

  procedure Clear (The_Family_Member : in out Family_Member_Type);
  -- Sets the Family member to the default values

  procedure Put (The_Information : in Information.Information_Type;
                 The_Family_Member : in out Family_Member_Type);
  -- Puts the given information into the family member

  function Info (The_Family_Member : in Family_Member_Type)
    return Information.Information_Type;
  -- returns the information associated with the family member

private

  type Family_Member_Record is
    record
      Info   : Information.Information_Type;
    end record;

  type Family_Member_Type is access Family_Member_Record;

end Family_Member;

-------------------------------------------------------------------
```

```
with Family_Member;
package Family_Tree2 is

  type Family_Tree_Type is private;

  procedure Put
             (The_Family_Member : in Family_Member.Family_Member_Type;
              The_Family_Tree   : in out Family_Tree_Type);
  -- puts the given family member onto the family tree

  procedure Sort_and_Report (The_Family_Tree : in Family_Tree_Type);
  -- Sorts the family tree and produces a report

private

  type Member_List_Record;
  type Member_List is access Member_List_Record;

  type Family_Tree_Type is
    record
      Current_Member : Member_List := null;
      First_Member   : Member_List := null;
    end record;

end Family_Tree2;

------------------------------------------------------------------

with Information;
with Family_Member;
with Family_Tree2;

procedure Make_Tree2 is
-- This is the driver procedure for the Family Tree version 2.

  The_Family_Member : Family_Member.Family_Member_Type;
  The_Family_Tree   : Family_Tree2.Family_Tree_Type;

begin
  -- open the input file
  Information.Open_Information_File;

  -- While there is more information, get the information, put it in
  -- the family member, and add the family member to the Family Tree.
  while not Information.End_of_File loop

    Family_Member.Clear (The_Family_Member => The_Family_Member);
    Family_Member.Put (The_Information => Information.Get,
                       The_Family_Member => The_Family_Member);
    Family_Tree2.Put (The_Family_Member => The_Family_Member,
                      The_Family_Tree   => The_Family_Tree);
  end loop;

-- process the information and produce the Family Tree report
  Family_Tree2.Sort_and_Report (The_Family_Tree => The_Family_Tree);

end Make_Tree2;

------------------------------------------------------------------
```

C. Specification for Example 3

```
generic
  type List_Element is private;

package Single_Unbounded_List is
  type Handle is private;
  type List is private;

  No_Element_Found   : exception;
  End_of_List        : exception;
  Not_At_End_of_List : exception;
```

```
      function Create_List return Handle;

      function Get_Element (At_Position : Handle)
         return List_Element;

      function Next(From : Handle) return Handle;

      procedure Append (New_Element : in list_element;
                        At_Position : in out handle);

      procedure Insert (New_Element : in List_Element;
                        At_Position : in out Handle);

      function At_End_of_List (This_List : in Handle)

      return boolean;

private
  type List is
    record
      Data : List_Element;
      Link : Handle;
    end record;
  type Handle is access List;

end Single_Unbounded_List;

------------------------------------------------------------------

-- package spec person_data

package person_data is
  type person_element is private;

  function make_list_element return person_element;

  procedure set_name (person : in out person_element;
                      name : string;
                      last : natural);

  function get_name (person : person_element)
    return string;

  procedure set_dob (person : in out person_element;
                     dob    : string;
                     last : natural);

  function get_dob (person : person_element)
    return string;

  procedure set_mother (person : in out person_element;
                        mothers_name : string;
                        last : natural);

  function get_mother (person : person_element)
    return string;

  procedure set_father (person : in out person_element;
                        fathers_name : string;
                        last : natural);

  function get_father (person : person_element)
    return string;

  function has_parent (person : person_element)
    return boolean;

  function element_is_printed (person : person_element)
    return boolean;

  procedure mark_person_as_printed
               (person : in out person_element);
```

```
private
  type person_element is
    record
      name : string(1..20);
      dob  : string(1..8);
      mother : string(1..20);
      father : string(1..20);
      is_printed : boolean;
    end record;

end person_data;

------------------------------------------------------------------

-- package instantiation for person_element_list

with single_unbounded_list;
with person_data;

package person_element_list is new
   single_unbounded_list (list_element => person_data.person_element);

------------------------------------------------------------------

-- package spec for Family Tree
-- postpone all with clauses to body, as
-- this is possible

package family_tree3 is

  procedure input_family_tree(from_file : string);
  -- Handle for the list is saved in a global
  -- variable in the package body

  procedure output_family_tree (to_file : string);
  -- Handle for the list is retrieved from global
  -- variable in the package body

end family_tree3;

------------------------------------------------------------------

-- Assuming that this is a single user program,
-- it is not designed with reentrant code, but on
-- the other hand, very few dependendencies are
-- created between the specs.  This is appropriate
-- for a prototyping mode, but might be redone
-- for a production version of the program.

with Family_Tree3;
procedure Make_Tree3 is
  data_file : constant string := "INPUT.TXT";

  output_File : constant string := "TREE3.OUT";
begin
  Family_Tree3.input_Family_Tree(Data_File);
  Family_Tree3.output_Family_Tree(Output_File);
end Make_Tree3;
```

Generic Target Acquisition Device

Chuck L. Carpenter

Telos Federal Systems
Lawton, Oklahoma

ABSTRACT

The Generic Target Acquisition Device (GTAD) is a communications device which has been designed and developed in Ada for the US Army. The primary objective of GTAD is to simulate the devices used in the Force Development Test and Experimentation (FDTE) by accepting messages addressed to GTAD and transmitting messages normally generated by target acquisition devices. Ada's powerful tasking mechanism was used extensively in the communications processing of GTAD to achieve a multi-tasking environment allowing the user to receive and transmit messages in the background while performing other system functions in the foreground. In addition to providing communications processing, GTAD provides a unique "authoring capability" which allows the user to create any message according to a predefined message syntax structure as well as transactions journals which monitor message traffic across the tactical nets.

INTRODUCTION

GTAD was designed in an effort to provide the US Army with a cost effective device that would simulate the message formats output by target acquisition devices that would not be present for the FDTE. GTAD's objectives include the automatic transmission and reception of messages through a sophisticated scenario driver, the monitoring of all message traffic across the tactical nets, and the unique ability for the user to generate new messages according to a predefined message syntax structure. The message syntax structure developed for GTAD had to be flexible enough to allow the user to change any existing message during FDTE as well as define entirely new messages for the test. The communications drivers for GTAD were designed to be as reliable as possible so that any messages processed by GTAD would not crash the system.

The use of Ada will allow GTAD to become a very sophisticated and reliable device. GTAD makes wide use of Ada's access, record, and array constructs to implement the message syntax structure. Because of the real time nature of GTAD's communications drivers, tasking was extensively used. This paper discusses the objectives and characteristics of GTAD's operational environment and concludes with a section on how Ada is used to achieve these objectives.

THE GTAD OPERATIONAL ENVIRONMENT

The code for the GTAD system is being developed on four desktop microcomputers. The target machines that are to be utilized for the FDTE are three portable microcomputers with special tactical modems. The tactical modems were designed on printed circuit boards to fit in the expansion slots of the microcomputers allowing GTAD to communicate with other tactical equipment utilizing the standard Army combat radio net.

COMMUNICATIONS PROCESSING

A powerful attribute of GTAD is its ability to simulate the message formats of many devices at the same time. This allows GTAD to be utilized in the FDTE as a single source for target acquisition devices. One GTAD may be used to replace all target acquisition devices that would normally have to be present at the FDTE. GTAD maintains a device table which indicates the devices GTAD is currently simulating. If a message transmitted on the net is addressed to one of the GTAD devices, then GTAD will respond with an appropriate response and place the message into one of its queues for user viewing.

The communications drivers of GTAD consist of six tasks which are activated whenever a message is either transmitted or received on one of the communications channels. The modem sends a special "end of message" code to indicate the end of the message on a received message. The receive task processes the message to determine if the message was sent from a valid subscriber and is correctly serialized. GTAD will respond to the subscriber immediately with either a positive or negative acknowledgement to the originator of the message. The processing of the message and the appropriate response to that message are completely transparent to the user of the GTAD system.

GTAD maintains three queues that are used for all message traffic. Messages that are transmitted manually by the user or automatically from a predefined scenario are placed into the transmit queue for that channel. Messages received with no errors from a subscriber are placed into the receive queue while messages received with errors are placed into the error queue with an appropriate error attached to the end of the message. The user may view or delete any message by selecting a message from the appropriate queue. The message may be viewed in a hex or octal dump format for debugging purposes or viewed as a normal ASCII character stream.

SCENARIO PROCESSING

One of the more sophisticated features of GTAD is the scenario driver which allows the user to automatically transmit and receive messages without any user intervention. The user creates a scenario which consists of predefined messages that are saved on the hard disk. The scenario file contains the names of messages which are stored as files on the hard disk as well as a delta time for each message. The delta time specifies the time for GTAD to wait between the transmission of each message. The scenario processor allows the user to start a scenario automatically at a specified time. The scenario file also contains a position number which refers to the current line number in the scenario file so that the user knows exactly where the scenario is executing at any given time. The user may also insert comments into the scenario file which will appear to the user during the execution of the scenario. The scenario file may also contain pause statements so that the scenario may be temporarily suspended until the user resumes scenario execution again.

MESSAGE SYNTAX STRUCTURE

GTAD provides a unique "authoring capability" which allows the user to create any message according to a predefined message syntax structure. This message syntax structure defines all the information needed to bit compress or expand a message for transmission or reception. One of the requirements for the GTAD system was the ability for the user to define or change any message structure without a recompilation of the system. As new tactical devices come into existence, the GTAD system can be used to simulate the message formats output by these devices. The proposed message outputs from these new devices may be completely tested before the actual tactical device is developed. This allows tactical devices that must interface with the new device to test their software before the device being developed is completed.

The message syntax structures are stored as ASCII files on the GTAD system allowing the user to create or modify any syntax file with the editor of his choice. The user may annotate comments into the syntax structure so that the syntax file is easier to modify and comprehend. Comments are enclosed between the left and right braces ("{ ... }"). The syntax structure not only defines the compression technique used to compress and expand the file, but also the display format that the user will see on the screen. The GTAD editor also uses this syntax structure to protect certain fields from the user and checks the validity of data entered into these fields as well. The syntax description also contains information on all the fields in the message. If the fields are mandatory for entry, then the GTAD editor will require the field to be filled out before the message can be transmitted. The syntax description contains information for the data type of each field and the number of bits to use to compress each field.

TRANSACTION JOURNAL PROCESSING

The transaction journals are used to monitor all messages transmitted and received on the tactical nets. GTAD maintains two transaction journals. The net transaction journal records all messages transmitted by any device on the net. The GTAD transaction journal is a subset of the net transaction journal and records only messages transmitted to GTAD. The transaction journals contain information such as the originator of the message, the date and time the message was received or transmitted, and the message category and type. Any errors associated with the message will be annotated to the side of the message.

GTAD DESIGN GOALS

This section discusses the design elements of Ada that were used to achieve the operational environment described above. These elements include tasking, access types, and exception handling. Of primary concern during the development of the GTAD system was that the target code would have to run on a portable microcomputer. Because of memory limitations and CPU speed, special attention was given in the use of tasking and access types.

TASKING

GTAD is comprised of seven tasks, six of which control the communications channels of the GTAD system. These tasks are the Channel 1 and 2 Receive Tasks, Channel 1 and 2 Transmit Tasks, Net Timer Task, Queue Task, and Clock Task.

The transmit and receive tasks are extremely time critical since the data is received and transmitted to the tactical modems at 4800 baud. Once a message is received by GTAD, the proper response must be sent to the originator of the message within 500 milliseconds. GTAD must then place the message into the appropriate queue. This processing of messages is completely transparent to the user of the GTAD system.

The queue task acts as a server task in that it is responsible for acting as a rendezvous point for the transmit and receive tasks. Since the two nets utilize the same receive queue, an error could occur if both receive tasks were to add a message to the receive queue at the same time. By utilizing the Ada rendezvous mechanism, one task will be blocked until the first task finishes adding the message to the receive queue.

The net timer task is also a server task which is called by the transmit and receive tasks to determine the duration of time that the net has been free. GTAD cannot transmit a message until the proper "time window" is available. This precludes other systems on the net from transmitting at the same time.

The clock task is an independent task which is used to update the real time clock at the bottom of the GTAD user screen.

The tasking mechanisms provided by Ada has significantly reduced the overall coding and design of the GTAD project. The background communications tasks of GTAD can process messages in the background while the user is left to execute other system functions in the foreground.

DYNAMIC ALLOCATION

Access types were extensively used by the GTAD system since a typical message syntax structure occupies as much as 70K of memory. The GTAD system image occupies 450K of memory. Since there is a great deal of data structures to deal with, almost all of the large data structures are allocated on the heap to conserve memory. To free memory, the data structures are immediately deallocated when no longer needed through the Ada mechanism of Unchecked Deallocation. Special attention was given when deallocating an object since there was no automatic garbage collection associated with our version of the compiler.

EXCEPTION HANDLING

Since GTAD is to be used as a test tool for testing electronic warfare equipment, it was imperative to design reliability in from the start. The use of Ada's exception handling has allowed GTAD to become a very reliable system. For example, if a new message is received by the communications handlers with a bit stream that GTAD can not recognize, an exception handler is invoked that will place the message into the error queue with an appropriate error message attached.

CONCLUSION

GTAD has become an extremely versatile and reliable test tool through the use of the Ada language. GTAD is a system that can be easily modified for future capabilities because of the package nature of Ada. The interface between new tactical systems may now be tested with existing systems by utilizing the capabilities of the GTAD message syntax structure.

The fact that Ada has built in support for tasking has allowed for the communications handlers of GTAD to process messages in the background while the user performs other system functions in the foreground. The total throughput of the system is further enhanced by utilizing the tasking mechanisms of Ada.

The extensive use of Ada's exception handlers has allowed GTAD to become a very reliable system pinpointing errors that can occur in new message formats that are introduced into the GTAD system.

CHUCK L. CARPENTER is a software engineer at Telos Federal Systems in Lawton Oklahoma. He received his B.S. in Computer Science from Kearney State College in 1984. At Telos Federal Systems, he has been involved in the design and coding of a debugger as well as real time communication drivers for the Battery Computer System. Currently he is involved in the development of the GTAD project.

ADVANCED ADA TASKING TECHNIQUES FOR MOTOR SIMULATION AND CONTROL

1LT Leonard S. Kim

University of California at Berkeley
Department of Mechanical Engineering

ABSTRACT

This project illustrates techniques for task organization and communication for motor control and simulation. A package containing tasks for motor plant simulation is derived. Controller packages utilizing simple proportional, integral, and derivative algorithms are then developed. A short example with a single plant and single controller is presented to illustrate the use of the two packages. In order to increase the number of tasks involved and monitor the behavior of these multiple tasks several modifications are made to the control package using advanced Ada tasking techniques. Applications for controller evaluation using these unique abilities in Ada are discussed.

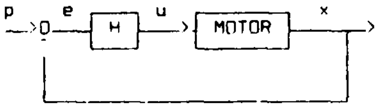
The programs presented in this paper serve to introduce and illustrate the rendezvous mechanism, scheduling, task activation, and task termination all within the context of DC motor control. All motor plant simulations rely on a matrix representation of the dynamic system which serves to demonstrate the advantages and limitations of Ada's matrix handling capabilities.

INTRODUCTION

Ada is unique among general purpose languages because all the constructs for real time event scheduling, multi-tasking and low level I/O are included within the syntax of the language[1]. It was designed with real time systems in mind. Modular construction is supported through the concept of an Ada Package where all logically related entities are encapsulated into one program unit. Ada's multi-tasking ability eliminates the need for non-portable hardware-dependent or operating system based real time schedulers.

The objective is control a series of DC motors running concurrently. One separate task unit is responsible for the control of each motor. In simulation, independent tasks model the dynamic behavior of these motors. In actual implementation, data acquisition and actuation software would replace these tasks. Controlling input signals and system feedback are exchanged via rendezvous. These examples illustrate many of the strong and weak points of the Ada tasking mechanism.

The overall system is represented below:



Position output (x) of each motor is fed back to calculate position error (e). The each controller task (H) uses this error to determine the new actuation (u). The each motor task takes this value, simulates dynamics of the motor and produces a new output. There are as many such systems operating in parallel as there are motors in the overall simulation.

MOTOR SIMULATION

The first objective is to create a single task type that serves as a digital simulation of the motor. The preferred method in Ada is to create task types embedded in packages rather than writing new tasks for each application. This approach which serves to reduce the amount of code written, implicitly requires that these task type definitions be flexible. Ada allows for the definition of access types to objects. Since a task type can be used to define an object, access variables to tasks are allowed. In addition, arrays of access variables can be created to represent arrays of concurrent tasks.

A more sophisticated approach to simulation might include a continuous model of the motor using some type of Runga-Kutta integration scheme, but for our purposes a discrete time model will be simpler.

A general representation of a continuous time DC motor model might be:

$$G(s) = \frac{Km}{s\ (Tm*s + 1)}$$

which defines the second order open loop transfer function. Km is referred to as the motor gain constant and Tm is time constant. Digital domain transfer is applied to the open loop equation assuming a zero-order-hold sampling of the system. The general representation of the digital model:

$$G(z) = \frac{b1*z + b2}{z**2 + a1*z + a2}$$

where a1, a2, b1, and b2 are referred to as the transfer function coefficients and are some function of the motor gain, time constant and sampling time. This transfer function can then be easily represented in the familiar difference equation format:

$$y(k+1) = b1 * u(k) + b2 * u(k-1) - a1 * y(k) - a2 * y(k-1)$$

where y(k) represents the motor position output at time step k and u(k) represent the input to the system at time step k. A more compact and powerful representation can be made by rewriting the coefficients and variables in vector notation:

$$y(k+1) = Thetat * Phi$$

where Thetat represents the transpose of a column vector of length n (n = 4 in this case) containing the motor plant coefficients a1, a2, b1, and b2. Phi is also a column vector of length n containing the values -y(k), -y(k-1), u(k), and u(k-1).

This formulation can be used to describe any n-th order linear system. It will be used here to perform a digital simulation of a motor plant. The specification of the desired task type is located in the specification of the package PLANT:

```
with MAT_FLOAT;use MAT_FLOAT;

package PLANT is

    task type MOTOR_TASK is
       entry PARAMETERS(T : in COL_POINTER);
       entry COMMAND(U : in  M_ELEMENT);
       entry SAMPLE( Y : out M_ELEMENT);
    end MOTOR_TASK;

    type MOTOR_POINTER is access MOTOR_TASK;
    type MOTOR_ARRAY is array(POSITIVE range <>)

                                         of MOTOR_POINTER;

end PLANT;
```

This package makes use of an external package MAT_FLOAT which is general purpose matrix manipulation package that allows access variables to refer to vectors and matrices. The types COL_POINTER, ROW_POINTER, and MATRIX_POINTER are made available in this package.

This package specification contains one task type definition MOTOR_TASK and two normal type definitions. Tasks of type MOTOR_TASK have three entry points each with an input or an output parameter: the first entry PARAMETERS receives a pointer to a column vector containing the coefficients of the plant transfer function (theta), the second entry COMMAND receives the motor input U(k), and the SAMPLE entry provides the motor output position Y(k). Recall that M_ELEMENT is a subtype of float.

The type MOTOR_POINTER is an access variable type to any object of type MOTOR_TYPE. Any variable of this type points to a task. Similarly the type MOTOR_ARRAY allows for the creation of a array of pointers to tasks. Any variable of this type points is an array containing a bank of pointers to concurrently executing MOTOR_TASKS.

```
package body PLANT is

   procedure UPDATE_PHI(PHI:in out COL_POINTER;
                        UK1,YK1:in M_ELEMENT) is

      ORDER : INTEGER;
      begin
         ORDER := PHI'LENGTH / 2;

         for I in 1..(ORDER-1) loop
            PHI(ORDER + 1 + I) := PHI(ORDER+I);
            PHI(I + 1)         := PHI(I);
         end loop;

         PHI(1) := -1.0 * Yk1;
         PHI(ORDER + 1) := Uk1;
      end UPDATE_PHI;

   task body MOTOR_TASK is
      --
      --  see below for task listing
      --
   end MOTOR_TASK;

end PLANT;
```

The body of the package PLANT contains a procedure definition and the body of the task type MOTOR_TYPE. Notice that the procedure is not mentioned in the specification of the package. The procedure UPDATE_PHI is intended to be used only by tasks of type MOTOR_TYPE. By placing it in the body of the package, this procedure is visible only to the members of the package following its definition. If a program were to with PLANT;use PLANT; it would gain access to the type definitions presented in the specification, but, the procedure UPDATE_PHI would remain hidden and unaccessible.

The column vector PHI defined above must be updated during each iteration of the simulation. The procedure takes three input values: PHI,Uk1, and Yk1, and returns the modified value of vector PHI. The pointer PHI inside the procedure is a different local version of the pointer PHI defined inside the task body. The use of a different name inside the procedure might clear this confusion but also would reduce the readability of the update process. The compiler should have no problem distinguishing the two.

```
task body MOTOR_TASK is
   THETA_TRANSPOSE : ROW_POINTER;
   PHI             : COL_POINTER;
   Yk1,Uk1 : M_ELEMENT := 0.0;

   begin
     accept PARAMETERS(T : in  COL_POINTER) do
       THETA_TRANSPOSE := TRANS(T);
     end PARAMETERS;
     PHI :=  new COL_VECTOR'(
               THETA_TRANSPOSE'RANGE =>  0.0);
     loop
       select
         accept SAMPLE(Y: out M_ELEMENT) do
           Y:= Yk1;
         end SAMPLE;
       or
         terminate; -- closed until master done
       end select;

       accept COMMAND( U : in M_ELEMENT) do
         Uk1 := U;
       end COMMAND;

       UPDATE_PHI(PHI,UK1,YK1);
       Yk1 :=  THETA_TRANSPOSE * PHI;
                           -- simulate process
     end loop;
  end MOTOR_TASK;
```

The body of the task type MOTOR_TYPE begins with a simple accept statement for plant parameters. After the start of execution, tasks of this type will suspend at the accept statement until rendezvous is accomplished. When a task or subprogram makes the entry call, a pointer to the plant coefficients is passed through the parameter T and copied to the task body local version of the pointer THETA_TRANSPOSE. Upon completion of rendezvous a vector to hold the contents of phi is created.

```
PHI := new COL_VECTOR'(
               THETA_TRANSPOSE'RANGE => 0.0);
```

Recall that PHI is a pointer to a column vector. new is an allocator which serves essentially the same roles as the functions malloc() or alloc() serve in the C language. An allocator creates an object and yields an access variable to that object. In this case the object, a column vector, is initialized to have all zeroes and has the same dimensions as the vector pointed to by THETA_TRANSPOSE. Now with a plant model and appropriately initialized vectors the simulation begins upon entering the loop.

Inside the loop is an example of a select statement with a terminate alternative:

```
  select
     accept SAMPLE( Y : out M_ELEMENT) do
        Y:= Yk1;
     end SAMPLE;
  or
     terminate;  --closed until master is done
  end select;
```

The task will suspend at the SAMPLE accept statement awaiting rendezvous on each iteration of the loop unless the terminate alternative is open. The terminate is open only if all dependent task are terminated or themselves waiting to terminate, and the master of the task is finished execution and waiting to terminate.

Next, the command input is received via rendezvous, PHI is updated, and finally to complete the iteration the plant is simulated one more time period using some very convenient notation.

Construction of a Control Type Task

A control task type PID_CONTROL_TASK is defined inside a new package CONTROL_PAC. Tasks of this type execute a simple-minded control algorithm using proportional, integral, and derivative error terms. The specification of the package contains the specification of the task:

```
with TEXT_IO,MAT_FLOAT,PLANT;
use TEXT_IO,MAT_FLOAT,PLANT;

package CONTROL_PAC is
   task type PID_CONTROL_TASK is
        entry MOTOR_NAME(M : in MOTOR_POINTER);
        entry GAINS(PID : in COL_POINTER);
        entry REFERENCE( R : in M_ELEMENT);
   end PID_CONTROL_TASK;

   type PID_POINTER is access PID_CONTROL_TASK;

end CONTROL_PAC;
```

The package makes use of definitions found in TEXT_IO, MAT_FLOAT and the new package PLANT. The task type PID_CONTROL_TYPE has three entry points: MOTOR_NAME to receive a pointer to the appropriate motor task, GAINS to receive controller gains, and finally REFERENCE to receive setpoint information from master tasks. The type PID_POINTER, like MOTOR_POINTER is an access variable to a task, in this case tasks of type PID_CONTROL_TASK.

```
package body CONTROL_PAC is

   task body PID_CONTROL_TASK is
      SET_POINT : M_ELEMENT := M_ELEMENT(0);
      MOTOR     : MOTOR_POINTER;
      Yk,Uk,Kp,Ki,Kd, ERROR : M_ELEMENT
                            := M_ELEMENT(0);
      LAST_ERROR, ERROR_SUM : M_ELEMENT
                            := M_ELEMENT(0);
      begin
         accept MOTOR_NAME(M : in
              MOTOR_POINTER) do
            MOTOR := M;
         end MOTOR_NAME;

         accept GAINS( PID : in COL_POINTER) do
            Kp := PID(1);
            Ki := PID(2);
            Kd := PID(3);
         end GAINS;
         loop
            select
               accept REFERENCE(R : in
                            M_ELEMENT) do
                 SET_POINT := R;
               end REFERENCE;
            else
               null;
            end select;

            MOTOR.SAMPLE(Yk);

            ERROR := SET_POINT - YK;
            Uk := Kp*ERROR +
                  Kd*(ERROR-LAST_ERROR)+
                  Ki*ERROR_SUM;
            MOTOR.COMMAND(Uk);

            ERROR_SUM  := ERROR_SUM + ERROR;
            LAST_ERROR := ERROR;
            delay 0.1;
         end loop;

      end PID_CONTROL_TASK;
end CONTROL_PAC;
```

The body of the task has two preliminary entries to receive the name of the motor to control and the appropriate gains to use. Inside the loop is another example of a select statement for the entry REFERENCE:

```
select
   accept REFERENCE(R : in M_ELEMENT) do
      SET_POINT := R; -- receive new  set point
   end REFERENCE;
else
   null;
end select;
```

In this case the else alternative is null. On each iteration, the task will check for any entry calls to REFERENCE. If none are in queue, the task will simple fly by this entry by selecting the else alternative which does nothing. The same effect could be achieved using a delay alternative with zero delay. In between issuing entry calls to sample motor data and to send motor command values, the task executes the control algorithm. At the end of each iteration, the task delays a finite period of time, 100ms in this case, as it would in real time implementation.

With these two packages, one can now write a simple procedure to simulate the control of one motor with one controller.

Single Motor Simulation

The procedure SIMU1 is a simple program that will act as master for our two tasks:

```
with MAT_FLOAT,PLANT,CONTROL_PAC;
use MAT_FLOAT,PLANT,CONTROL_PAC;

procedure SIMU1 is
  MOTOR_A : MOTOR_POINTER; -- to motor task
  CONTROL : PID_POINTER;   -- to control task
  TRANSFER_Fcn :  COL_POINTER := new
       COL_VECTOR'(-1.66,0.6,0.002,0.001);
  GAINS_VEC : COL_POINTER := new
       COL_VECTOR'(150.0,10.0,140.0);
begin
  MOTOR_A := new MOTOR_TASK; -- make motor task
  CONTROL := new PID_CONTROL_TASK;
                              -- make control task
  MOTOR_A.PARAMETERS(TRANSFER_Fcn);
                             -- send transfer fcn.
  CONTROL.MOTOR_NAME(MOTOR_A);
                             -- send motor_name
  CONTROL.GAINS(GAINS_VEC); -- send gains
  CONTROL.REFERENCE(0.1); -- give it a step
  delay 30.0;  -- simulate for 30 seconds
  abort CONTROL.all,MOTOR_A.all;
end SIMU1;
```

Two access variables MOTOR_A and CONTROL are first defined to represent the two operating tasks. Vectors containing transfer function coefficients and controller gains are declared and created, initialized and assigned to appropriately named access variables.

As the procedure begins, two tasks are created and begin execution. Recall that both task types immediately suspend execution awaiting some type of information. The first entry in MOTOR_A, PARAMETERS, is called and transfer function coefficients are passed. The task MOTOR_A would then continue execution entering its real time loop. The next two entry calls to CONTROL pass the name of the motor and the control gains vector via rendezvous with that task. At this point both tasks, conceptually, are running through their simulation loops.

CONTROL.REFERENCE is called and passed a setpoint of 0.1 during rendezvous. SIMU1 then suspends for 30 seconds while the motor and control tasks execute. If desired, PUT statements could be placed inside the control task to monitor input values and response. Finally the abort command forces a termination of the two tasks before the procedure ends execution.

Multiple Motor and Control Simulation

In order to increase the number of tasks involved and monitor the behavior of these multiple tasks several modifications are made to the control package which is now called CONTROL_PAC2. First two type definitions are added to the package specification. ID_TYPE is an integer subtype ranging between 1 and 9. This will be used to identify nameless, concurrently executing tasks and place system information on the screen". The type CONTROL_ARRAY is defined to hold a series of pointers to control tasks.

Finally the entry MOTOR_NAME is modified to accept an id number in addition to a motor task pointer. The package specification:

```
with TEXT_IO,MAT_FLOAT,PLANT;
use TEXT_IO,MAT_FLOAT,PLANT;

package CONTROL_PAC2 is
   subtype ID_TYPE is INTEGER range 1..9;
    task type PID_CONTROL_TASK is
       entry MOTOR_NAME(M : in  MOTOR_POINTER;
                        ID :  in ID_TYPE);
      entry GAINS(PID : in COL_POINTER);
       entry REFERENCE( R : in M_ELEMENT);
   end PID_CONTROL_TASK;

   type PID_POINTER is access PID_CONTROL_TASK;
   type CONTROL_ARRAY  is array(POSITIVE
                        range <>) of PID_POINTER;
end CONTROL_PAC2;
```

The body of CONTROL_PAC2 is essentially the same as CONTROL_PAC except for a few enhancements that will allow individual control tasks to be monitored during execution. ANSI full screen terminal displays are used so that outputs can be sent to any position on the screen. For example *"[x;1H" will place the* cursor on the x-th row and first column. This will work on VT-100 compatible terminals or on an IBM PC compatible with ANSI.SYS placed in the CONFIG.SYS file.

```
package body CONTROL_PAC2 is

 task body PID_CONTROL_TASK is

   -- all previous variable definitions --
   -- in CONTROL_PAC are repeated here  --

   ID_NUM : ID_TYPE;
   SCREEN_ROW : STRING(1..6)
                    := ASCII.ESC & "[x;1H";
   package IO_FLOAT is new FLOAT_IO(M_ELEMENT);
   use IO_FLOAT;

   begin
      accept MOTOR_NAME(M : in MOTOR_POINTER;
                           ID : in ID_TYPE) do
         MOTOR := M;
         ID_NUM := ID;
      end MOTOR_NAME;

      SCREEN_ROW(3) := CHARACTER'VAL(
          CHARACTER'POS('0')+ID_NUM);
      -- GAINS entry same as before --
      loop
         -- REFERENCE entry same as before --
         -- control calculations are same --
         MOTOR.COMMAND(Uk); -- send uk

         PUT(SCREEN_ROW &  "TASK " &
                INTEGER'IMAGE(ID_NUM) & " ");
         PUT(ITEM =>Yk, FORE =>5, EXP =>0);
         new_line;
         -- error calculation is same --
         delay 0.1;
      end loop;
   end PID_CONTROL_TASK;
end CONTROC_PAC2;
```

A string type variable SCREEN_ROW is created so that the id number of each task once received, can be loaded into the proper ASCII escape sequence. The language defined attribute/functions CHARACTER'VAL and CHARACTER'POS are used to load the correct character representation of the id number into SCREEN_ROW. The statement:

```
    PUT(SCREEN_ROW &  "TASK "  &
INTEGER'IMAGE(ID_NUM) & " ");
```

places the word "TASK" followed by the ID number of the task on the screen row that is the id number. Following that a PUT defined in IO_FLOAT is executed to report the position output of the control task's motor.

A new procedure SIMU2 is present to demonstrate the use of this new package. SIMU2 will dynamically create five motor task and five controllers while providing each of the ten tasks starting information on the fly:

```
with TEXT_IO,MAT_FLOAT,PLANT,CONTROL_PAC2;
use  TEXT_IO,MAT_FLOAT,PLANT,CONTROL_PAC2;

procedure SIMU2 is -- multiple task simulation

   subtype MOTOR_NUMBERS is INTEGER range 1..5;
   -- 5 motors and controllers
   PROCESS : MOTOR_ARRAY(MOTOR_NUMBERS);
   CONTROL : CONTROL_ARRAY(MOTOR_NUMBERS);
   TRANSFER_ARRAY :
     array(MOTOR_NUMBERS) of COL_POINTER :=(
     new COL_VECTOR'(-1.66,0.6,0.002,0.001),
     new COL_VECTOR'(-1.66,0.6,0.002,0.001),
     new COL_VECTOR'(-1.66,0.6,0.002,0.001),
     new COL_VECTOR'(-1.66,0.6,0.002,0.001),
     new COL_VECTOR'(-1.66,0.6,0.002,0.001));

   GAIN_ARRAY :
     array(MOTOR_NUMBERS) of COL_POINTER :=(
     new COL_VECTOR'(150.0,10.0,140.0),
     new COL_VECTOR'(100.0, 5.0, 98.0),
     new COL_VECTOR'( 80.0, 8.0, 50.0),
     new COL_VECTOR'( 10.0, 0.1,  9.0),
     new COL_VECTOR'(  1.0, 0.01, 0.9));

   CLEAR_SCREEN :
     STRING(1..4) := ASCII.ESC & "[2J";

begin

  PUT_LINE(CLEAR_SCREEN);
  for I in MOTOR_NUMBERS loop
    PROCESS(I) := new MOTOR_TASK;
                         -- create tasks
    CONTROL(I) := new PID_CONTROL_TASK;

    PROCESS(I).PARAMETERS(TRANSFER_ARRAY(I));
                         -- send tr fcn
    CONTROL(I).MOTOR_NAME(PROCESS(I),I);
                         -- send name & id
    CONTROL(I).GAINS(GAIN_ARRAY(I));
                         -- send cntrl gains
    CONTROL(I).REFERENCE(0.1);
                         -- set reference
 end loop;

 delay 30.0;   -- simulate for 30 seconds

   for I in MOTOR_NUMBERS loop
      abort CONTROL(I).all,PROCESS(I).all;
end loop;
   PUT_LINE(CLEAR_SCREEN);  -- clear screen
end  SIMU2;
```

The arrays PROCESS and CONTROL are defined to be of type MOTOR_ARRAY and CONTROL_ARRAY respectively. Two new variables TRANSFER_ARRAY and GAIN_ARRAY are arrays of pointers to vectors containing transfer function and control gain information respectively. In this case the same motor transfer function is entered for each task. This will allow for a nice comparison of control performance. Different transfer functions describing several different systems of differing order could just as easily been used. The gain array contains 5 sets of PID gains chosen for comparison.

As execution begins an ASCII sequence is used to clear the screen. Then five iterations of creating control and motor task are carried out. Each control task is provided with a motor name and motor id number in addition to controller gains. Each motor/controller system is given a 0.1 step input. The simulation continues for 30.0 seconds. The screen output would appear something like this:

```
TASK  1   0.10000
TASK  2   0.10020
TASK  3   0.17233
TASK  4  24.03340
TASK  5  88.15968
```

where the number following the task number is the motor position. All five motors run to the end of the 30.0 second delay period then are killed using the abort command.

Conclusions

Use of the Ada language in control applications of real time systems was surveyed in depth. The language in general is extremely well suited for such systems and carries an inherent propensity to enforce good software engineering principles such as modular programming, encapsulation of data, stringent typing and information hiding. A reduced amount of time was spent debugging programs because entities are so carefully structured. The Ada rendezvous represents a significant improvement over standard real time operating system constructs because it is much easier to use and less prone to mistakes. These effects can combine to produce a software product that is reasonably dependable, maintainable, and reuseable over time.

The designers of Ada have created a language for large-scale and real time embedded systems. I have tried to show that for low level control applications, this high level language is extremely well equipped. The power of flexible object types, the overloading of operators for matrix manipulation and definition of real time tasking constructs within the syntax of the language make Ada an extremely attractive choice for large real time control systems.

References

Aho, Alfred V., Hopcraft, John E., and Ullman, Jeffery D. The design and Analysis of Computer Algorithms, Addison-Wesley Publishing Co., Reading, MA, 1974.

Astrom, Karl J., and Wittenmark, Bjorn. Computer Controlled Systems: Theory & Design. Prentice-Hall, Inc., Englewood Cliffs, N.J. 1984.

Auslander, David M. and Tham, C.H. "Real Time Programming and Motor Control - ME 230 Class Notes." Unpublished,1985.

Barnes, J.P.G., An Overview of Ada. Software-Practice and Experience, V10, pp.851-887, 1980.

Barnes, J.P.G., Programming in Ada, Addison-Wesley Publishing Co., London, 1982.

Bergman, Bruce, et. al. "Ada Compilers: Mission - Critical Software for the PC." in Computer Language, 3:12 and 4:1, December 1986, January 1987, Parts I & II.

Booch, Grady, Software Engineering with Ada, Benjamin/Cummings Publishing Co., Menlo Park, CA., 1983.

Burns, Alan, Concurrent Programming in Ada, Cambridge University Press, Cambridge, 1985.

Condiz, Marin. "Packages: Ada's Integrated Circuit." in Computer Language, 3:12, December 1986.

Gehani, Narain, Ada, An Advanced Introduction, Prentice-Hall Inc., Englewood Cliff, N.J., 1983.

Kim, Leonard. "Influence Diagram Based Expert System Tool Developed in Ada." unpublished, 1987.

Norton, Peter, Programmer's Guide to the IBM PC, Microsoft Press, Bellevue, WA., 1985.

Reference Manual for the Ada Programming Language, U.S. Department of Defense, January 1983.

Tomizuka, Masyoshi. "Advanced Control Systems-ME 232 Class Notes." Unpublished 1986.

1LT Leonard S. Kim has a BS in Operations Research from Princeton University, a MS in Mechanical Engineering from U.C. Berkeley, and a M.Eng. in the area of control theory also from U.C. Berkeley. In 1985, he was selected to participate in the Army's Technological Enrichment Program. Under the auspices of that program, LT Kim studied robotics and Ada as a graduate student at Berkeley for two years.

LT Kim is presently en route to Korea for a one year tour of duty with the 2nd Infantry Division.

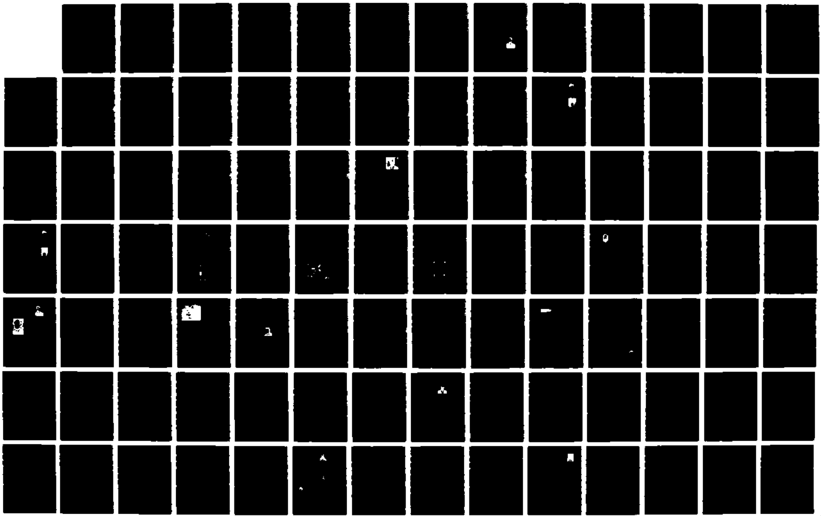
Endnotes

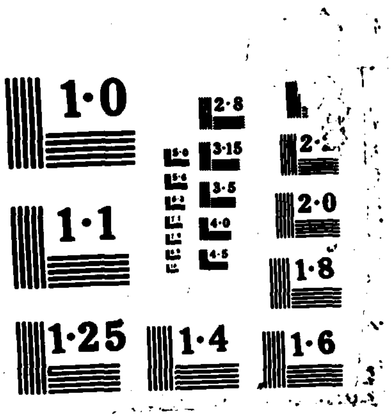
1.Other high level languages such as MODULA2 also have these types of features, however, no other language can boast the same level of enforced standardization.

2.This method of task identification and screen control was presented in "The Road to Ada Tasking" by Avram Tetewsky, **COMPUTER LANGUAGE**, Vol. 4, #8, August 1987.

AD-A190 936 PROCEEDINGS OF THE NATIONAL CONFERENCE ON ADA (TRADE MARK) TECHNOLOGY (6TH) HELD ON 14-18 MARCH 1988 IN ARLINGTON VA(U) NORFOLK STATE UNIV VA MAR 88 4/7

UNCLASSIFIED F/G 12/5 NL




1·0
2·8
3·15
3·5
4·0
4·5
2·2
2·0
1·1
1·8
1·25
1·4
1·6

Adaptive Robotic Control in Ada

1LT Leonard S. Kim

Department of Mechanical Engineering
University of California at Berkeley

ABSTRACT

This paper covers the development of an advanced robotic controller written in Ada. The objective of the project is to evaluate the applicability of Ada for complex control systems. Most of the advanced features of the Ada tasking model are utilized to illustrate their power and limitations. Controllers are implemented in Ada using the same convenient matrix notation commonly used to development the algorithms. Three concurrent task types are developed to handle plant simulation, system identification and actuator control. They provide good examples of an Ada tasking program in a practical application. Use of the Ada rendezvous mechanism and some of its limitations are addressed. The results of simulation as well as the details of system organization are presented.

INTRODUCTION

Control of robotic manipulators at the joint level is especially difficult because systems often encounter changing inertial loads due to manipulator configuration and payload changes. Classical control approaches with stationary control gains are accurate only inside limited load ranges. Adaptive controllers attempt to vary these feedback control gains in real time based upon estimates of the present configuration of the system. The Ada tasking model is shown to be extremely useful for these types of complex robotic control schemes.

A self-tuning controller is the most intuitive approach to adaptive control. It involves the use of an identifier to make estimates of plant parameters in real time. Using these estimates any one of a number of standard controller designs can be used to pick the gains. In this case a standard pole placement controller is tuned in real time based upon the most current estimates of the plant. This approach to control is useful because standard controllers are usually based on one model. In cases where the parameters of the model may vary significantly, such as large mechanical systems that encounter changing loads, these standard controllers may not perform acceptably.

This scheme is a particularly attractive Ada tasking application because the roles of identification and control are completely separate. Applications in which the partitioning of tasks is clearly defined are ideal for multi-processing environments. By developing applications in Ada, the benefits of multi-processing systems, when they become affordable, will be easily reaped without changes to the software. An example of such a system is a robotic manipulator where the partitioning of tasks is well defined at both the system level and the joint level.

The PLANT

The plant model formulation is based on the familiar difference equation format. For a linear second order discrete time system:

```
y(k+1) = b1*u(k)+b2*u(k-1)
         -a1*y(k)-a2*y(k-1)
```

where y(k) and u(k) represent system outputs and inputs respectively at time step k. a1, a2, b1 and b2 are coefficients to the system transfer function. Our formulation is given in vector notation:

```
Yk1 = Thetat * Phi
```

where Yk1 is the system output at discrete time step k + 1, Thetat is the transpose of a column vector of length n containing the coefficients of the discrete time transfer function, and Phi is a column vector of length n containing present and previous system input and output values.

A motor task defined in a package **PLANT** will be used to simulate the motor. Two select accept statements are used inside this task; one at the beginning of the task body to receive the initial system transfer function and one placed inside the simulation loop so that plant parameters can be changed during operation.

Ada allows more than one accept statement for each entry. The first entry **PARAMETERS** serves to hold the task at the beginning of execution until it receives a plant. The second select statement has a null alternative meaning that the task will simply fly by this statement unless some other task wants to change the parameters.

IDENTIFICATION

The objective of identification is to come up with an estimate of Theta, call it Theta_hat, using the information contained in Phi, without knowing Theta a priori. Hence the objective is to determine an estimate of the plant behavior:

```
 Yk1_hat = TRANS(Theta_hat) * Phi
```

such that the error:

```
    error = Yk1 - Yk1_hat
```

is driven to zero. In this application a recursive form of a least squares regression will be used to drive the error to zero. The purpose of this section is present a good tasking control example using a complex control algorithm. The theoretical basis for this form of control is extremely involved and requires separate treatment [Landau, 1979]. For our purposes suffice to say that the least square estimate uses the derivative of the summation of the squared prediction error. By setting the partial derivative of this sum to zero a formula for the estimate, Theta_hat, can be derived using only the error, Phi and values of the previous estimates. One consequence of this approach is that the estimate gains converge exponentially to zero. In situations where the plant parameters are time varying, forgetting factors are introduced into the formulation to prevent the estimate gains from converging to zero. The final formulation for the new estimates:

```
Theta_hat(k+1) :=   Theta(k) + F(k) *
             Phi(k) *  Error(k+1);

     where  F(k) := 1/L1 *

  (F(k-1)-(F(k-1)*P(k)*T(P(k))*F(k-1)))
-------------------------------------
    L1/L2 + T(P(k)) * F(k-1) * P(k)
```

using P for Phi and T for the transpose operator in the last equation. Lamda1 (L1) and Lamda2 (L2) represent forgetting factors. F is an nxn matrix referred to as the adaptation gain matrix. For the purposes of this paper it is only important to understand that the theory is developed using this sort of notation. The overloading of operators for matrices and vectors in Ada allows the control engineer to preserve the structure of the theoretical derivation minimizing mistakes and significantly speeding up the coding process. The same algorithm as it might be done in Ada:

```
NUMERATOR := F - (F*PHI*TRANS(PHI)*F);

DENOMINATOR := LAMDA1_OVER_LAMDA2 +
           TRANS(PHI) * F * PHI;

F := ONE_OVER_LAMDA1 *
                NUMERATOR/DENOMINATOR;

THETA_HAT := THETA_HAT +  F*PHI*Error;
```

where LAMDA1_OVER_LAMDA2 and ONE_OVER_LAMDA1 are predefined constants.

On the transient, it is entirely possible that the value for denominator becomes zero. This provides an excellent application to display use of an local block statement with an exception handler. When division by zero occurs, the language raises a NUMERIC_ERROR - one of the several predefined language exceptions. This error can easily be handled inside a local block:

```
begin
  F := ONE_OVER_LAMDA1 *
                NUMERATOR/DENOMINATOR;
  exception
    when NUMERIC_ERROR =>
      F :=  ONE_OVER_LAMDA1 *
                NUMERATOR/0.0001;
end;
```

The NUMERIC_ERROR exception is handled inside the block by replacing the denominator with a very small number. Ada exception rules state that when an exception occurs control can be transferred to a user-provided exception handler at the end of a block statement or program unit. If no handler exists, specific rules govern how that error will be propagated to upper level program units.

With this background, an identifier task type can be presented inside the package IDENTIFICATION:

```
with PLANT,MAT_FLOAT;
use PLANT,MAT_FLOAT;
package IDENTIFICATION is

   task type IDENTIFIER is

    entry SYSTEM_ORDER(DEGREE :
                          in INTEGER);
    entry DATA_IN(  CONTROL_INPUT,
         PLANT_OUTPUT : in M_ELEMENT);
    entry PARAMETERS_OUT(
    PLANT_PARAMETERS :
                       out COL_POINTER);
   end IDENTIFIER;
end IDENTIFICATION;

package body IDENTIFICATION is

  BETA : constant M_ELEMENT := 1000.0;
  LAMDA1 : constant M_ELEMENT := 0.9;
  LAMDA2 : constant M_ELEMENT := 1.0;
  ONE_OVER_LAMDA1 :
     constant M_ELEMENT := 1.0/LAMDA1;
  LAMDA1_OVER_LAMDA2 : constant
  M_ELEMENT := LAMDA1/LAMDA2;
  task body IDENTIFIER is separate;
end IDENTIFICATION;
```

This package describes a task type that is a generalized identification routine for a linear system of any order. Note that the task type has three entry points: one for the order of the system to be identified; one for the system inputs and outputs; and a third for the estimated parameters to be passed out. Inside the package body, several constants for the identification are defined. The task body is contained in a separately compiled unit as indicated by the statement:

```
task body IDENTIFIER is separate;
```

The body of the task type IDENTIFIER:

```
separate(IDENTIFICATION)

task body IDENTIFIER is
   PHI              : COL_POINTER;
   THETA_HAT        : COL_POINTER;
   F                : MATRIX_POINTER;
   TEMP             : ROW_POINTER;
   DENOMINATOR       : M_ELEMENT := 0.0;
   ERROR             : M_ELEMENT := 0.0;
   Yk1,Yk1_hat,Uk1   : M_ELEMENT := 0.0;
   N                 : INTEGER;

begin
  accept SYSTEM_ORDER(DEGREE  : in
                       INTEGER) do
    N := 2 * DEGREE;
  end SYSTEM_ORDER;
  F := BETA * IDENTITY(N);
  PHI := new COL_VECTOR'(1..N => 0.0);
  THETA_HAT :=  new COL_VECTOR'(1..N =>
                                0.0);
loop
    UPDATE_PHI(PHI,Uk1,Yk1);
    select
      accept DATA_IN(  CONTROL_INPUT,
        PLANT_OUTPUT : in M_ELEMENT) do
          Uk1  := CONTROL_INPUT;
          YK1      := PLANT_OUTPUT;
        end DATA_IN;
    or
      terminate;
    end select;

    Yk1_hat :=  TRANS(THETA_HAT) * PHI;
    ERROR := Yk1 - Yk1_hat;
    TEMP := TRANS(PHI) * F;
    DENOMINATOR :=  LAMDA1_OVER_LAMDA2
                        + TEMP * PHI;
    begin  -- watch division by zero
       F :=  ONE_OVER_LAMDA1*(F-(F
              *PHI*TEMP)/DENOMINATOR);
       exception
          when NUMERIC_ERROR =>
             F :=  ONE_OVER_LAMDA1*(F-
               (F*PHI*TEMP)/0.00001);
    end;
    THETA_HAT := THETA_HAT+F*PHI*ERROR;

    accept PARAMETERS_OUT(
                     PLANT_PARAMETERS :
                    out COL_POINTER) do
       PLANT_PARAMETERS := THETA_HAT;
    end PARAMETERS_OUT;
  end loop;
end IDENTIFIER;
```

The actual implementation of the least squares follows almost exactly from the example presented above. Slight modifications have been added to reduce duplication of terms and improve real time performance.

CONTROLLER

The control task makes use of a standard pole placement algorithm. The objective in such an approach is to achieve a certain type of dynamic response by dictating the poles of the closed loop system. Using these desired poles and an estimate of the plant provided by IDENTIFIER the appropriate feedback gains can be calculated. For simplicity a plant with stable zeros is chosen so that issues dealing with unstable pole-zero cancellation can be foregone. In actual implementation these stability issues must be dealt with more rigorously. The control task type is similar to the task types presented earlier:

```
with
TEXT_IO,MAT_FLOAT,PLANT,IDENTIFICATION;
use
TEXT_IO,MAT_FLOAT,PLANT,IDENTIFICATION;

package CONTROL_PAC3 is
```

```
 task type CONTROLLER_TASK is
   entry MOTOR_NAME(M : in
                  MOTOR_POINTER);
   entry GAINS(POLES :in ROW_POINTER);
   entry REFERENCE( R : in M_ELEMENT);
 end CONTROLLER_TASK;

 type CONTROLLER_POINTER is access
                  CONTROLLER_TASK;
end CONTROL_PAC3;
```

Note that the entry GAINS accepts coefficients of the characteristic equation that defines the desired closed loop poles. The body of the package CONTROL_PAC3 and the task type CONTROLLER_TASK are provided at the end of this section. Consider now the main procedure that utilizes these packages and actually conducts the simulation.

SIMULATION

The main procedure is SIMU3:

```
with
CALENDAR,MAT_FLOAT,PLANT,CONTROL_PAC3;
use
CALENDAR,MAT_FLOAT,PLANT,CONTROL_PAC3;

procedure SIMU3 is
   ORDER : constant INTEGER := 2;
   N     : constant INTEGER := 4;
   PROCESS :  MOTOR_POINTER := new
                 MOTOR_TASK;
   CONTROLLER : CONTROLLER_POINTER :=
           new CONTROLLER_TASK;
   D     : ROW_POINTER := new
       ROW_VECTOR'(1.0,-1.066,0.3688);
   -- pole placement transfer fnc.

   -- system transfer functions
   NO_LOAD      : COL_POINTER := new
    COL_VECTOR'(-1.6,0.6,0.002,0.001);
   WITH_LOAD   : COL_POINTER := new
    COL_VECTOR'(-1.8,0.8,0.003,0.002);

   begin
     PROCESS.PARAMETERS(NO_LOAD);
     CONTROLLER.MOTOR_NAME(PROCESS);
     CONTROLLER.GAINS(POLES =>  D);
     CONTROLLER.REFERENCE(1.5);

     delay 15.0;

     PROCESS.PARAMETERS(WITH_LOAD);
     CONTROLLER.REFERENCE(0.0);

     delay 15.0;

     PROCESS.PARAMETERS(NO_LOAD);
     CONTROLLER.REFERENCE(1.0);

     delay 15.0;
     abort CONTROLLER.all,PROCESS.all;
 end SIMU3;
```

This procedure creates the control and plant tasks (the identifier task is created inside the controller), and initializes all the system parameters. The system is given a step input of 1.5. Then after 15.0 seconds a new transfer function is injected along with a reference position value of 0.0. Finally the old transfer function is restored and a reference of 1.0 is requested.

In a real application this might correspond to a robot picking up a load, moving to another position, setting it down and then moving to another position. The controller is sent discrete time poles that correspond to continuous poles at 10 +/ 10j discretized at 0.05 seconds. The job of the identifier is to determine the transfer function parameters each time one is injected. The control has the responsibility of regulating the system about the reference point.

The results are shown in the following graphs. Notice that control response is well behaved despite the changes in plant. This means that the desired closed loop system is at work. The parameters a1 and a2 converge nicely to their respective values exponentially. Once there they lock on to their values and hold until the plant is changed again. The parameters b1 and b2 are not as well behaved but do eventually lock onto the correct values. The wild transient behavior of these parameter is not as significant when the range of actual values is considered. Of course this is simulation, results in a stochastic environment would not be as impressive. The controlling input contains slight oscillations but is well behaved overall.

For completeness the main loop of the controller is shown below:

```
package body CONTROL_PAC3 is
  task body CONTROLLER_TASK is
    ORDER :  constant INTEGER := 2;
    THETA_HAT  : COL_POINTER;
    Yk :COL_POINTER := new
      COL_VECTOR'(1..ORDER => 0.0);
    Ym :COL_POINTER  := new COL_VECTOR'
             (1..(ORDER+1)  => 0.0);
    D : ROW_POINTER;
    a1,a2,b1,b2 : M_ELEMENT := 0.0;
    NUMERATOR    : M_ELEMENT := 0.0;
    Yk1,Uk1      : M_ELEMENT := 0.0;
    IDENT        : IDENTIFIER;
    MOTOR        : MOTOR_POINTER;
  begin
    accept MOTOR_NAME(M  : in
            MOTOR_POINTER) do
      MOTOR := M;
    end MOTOR_NAME;

    accept GAINS(POLES  : in
                 ROW_POINTER) do
      D := POLES;
    end GAINS;
    IDENT.SYSTEM_ORDER(ORDER);
    loop
      select
        accept REFERENCE( R : in
                     M_ELEMENT) do
          Ym.all :=  (Ym'RANGE => R);
         end REFERENCE;
      else
          null;
      end select;

      IDENT.DATA_IN(Uk1,Yk1);
      delay 0.05; -- sample delay
      IDENT.PARAMETERS_OUT(THETA_HAT);

      a1 := THETA_HAT(1);
      a2 := THETA_HAT(2);
      b1 := THETA_HAT(3);
      b2 := THETA_HAT(4);
      MOTOR.SAMPLE(Yk1);

      Yk(2) := Yk(1);
      Yk(1) := Yk1;

      NUMERATOR := D*Ym-(D(2)-a1)*
       Yk(1)-(D(3)-a2)*Yk(2)- b2*Uk1;

      begin    -- check for b1 = 0.0
        Uk1  := NUMERATOR / b1;
      exception
        when NUMERIC_ERROR  =>
          Uk1 := NUMERATOR / 0.0001;
      end;

      if UK1 > 10.0 then UK1 := 10.0;
      elsif UK1 < -10.0
                    then UK1 := -10.0;
      end if;
     MOTOR.COMMAND(Uk1);
    end loop;
  end CONTROLLER_TASK;
end CONTROL_PAC3;
```

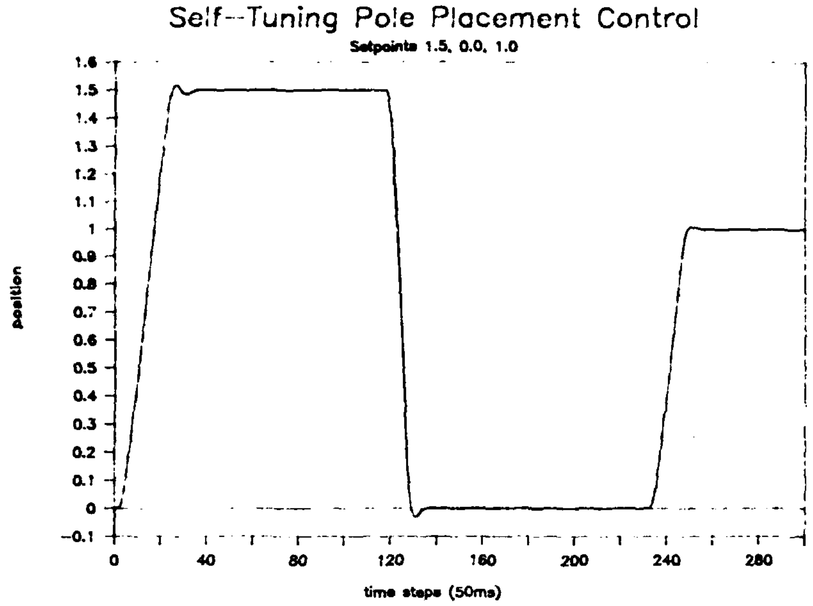
CONCLUSIONS

An adaptive self tuning controller useful for robotic applications was presented. The exercise has shown that Ada is extremely suitable for complex control algorithms. The overloading of operators for matrices and vectors allows engineers to preserve the structure of the theoretical derivation minimizing mistakes and significantly speeding up the coding process.

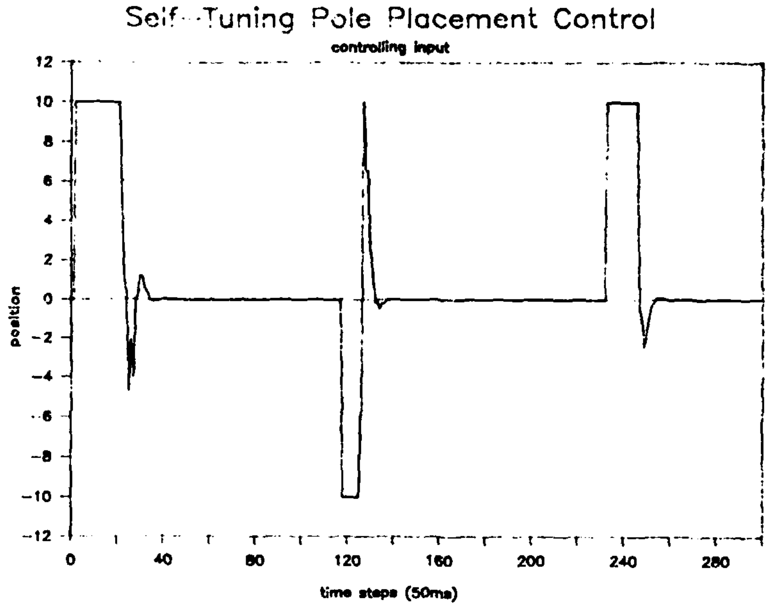
From a tasking standpoint, the Ada rendezvous represents a real improvement over standard real time operating system constructs because it is much easier to use and less prone to mistakes. These effects can combine to produce a software product that is reasonably dependable, maintainable, and reuseable over time.

The language does have several drawbacks. One is the lack of any mechanism to provide for quick task context switching. Slices, although defined within the language as one-dimensional arrays formed as a sequence of consecutive components of another one-dimensional array slices, do not allow for the use of slices within arithmetic expressions. This severely limits the usefulness of the convenient matrix and vector notation possible within the language since ranges within an array can not be used. Strong typing for the programmer is a mixed blessing that sometimes saves a program, but, often serves as nuisance. There is no ability to define procedure types as there is in MODULA2. Ada programs tend to be larger than similar programs developed in other languages. Finally the large size and the complexities of the language make Ada compilers extremely slow and expensive at the present time.

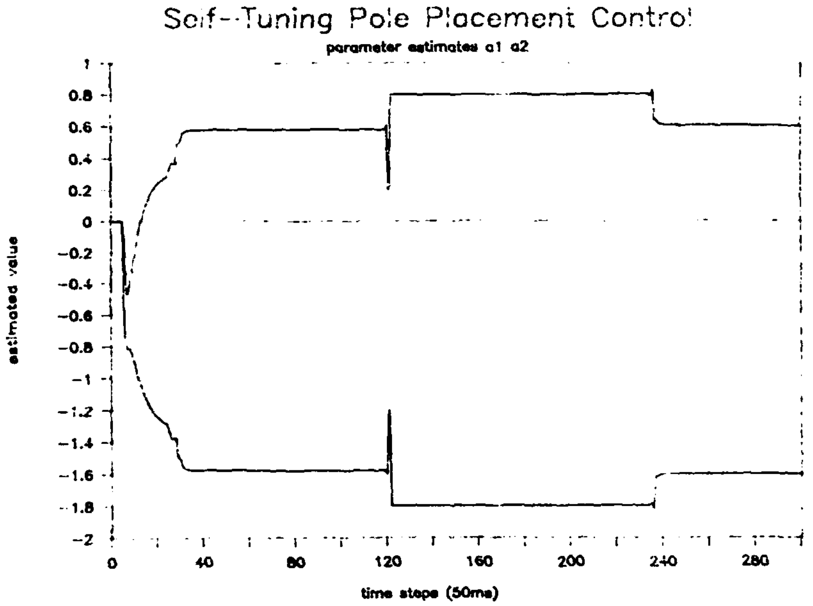
Despite this long list of complaints, many of which have quick fixes or will be addressed in time, the designers of the language seem to have achieved what they set out to do: design a language for large-scale and real time embedded systems. I have tried to show that for complex control applications, this high level language is extremely well equipped. The power of flexible object types, the overloading of operators for matrix manipulation and definition of real time tasking constructs within the syntax of the language make Ada an extremely attractive choice for large real time control systems.


Self--Tuning Pole Placement Control

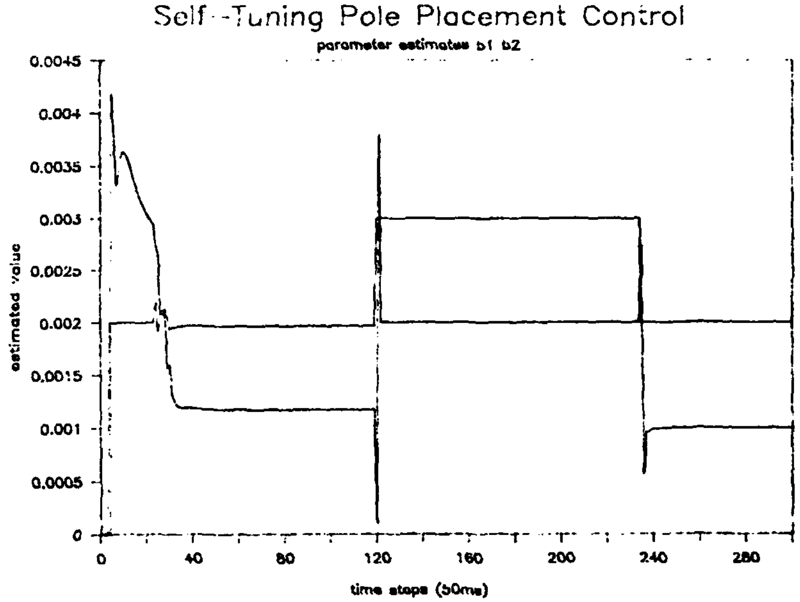
Setpoints 1.5, 0.0, 1.0


Self--Tuning Pole Placement Control

controlling input


Self--Tuning Pole Placement Control

parameter estimates a1 a2


Self--Tuning Pole Placement Control

parameter estimates b1 b2

References

Aho, Alfred V., Hopcraft, John E., and Ullman, Jeffery D. The design and Analysis of Computer Algorithms, Addison-Wesley Publishing Co., Reading, MA, 1974.

Astrom, Karl J., and Wittenmark, Bjorn. Computer Controlled Systems: Theory & Design. Prentice-Hall, Inc., Englewood Cliffs, N.J. 1984.

Auslander, David M. and Tham, C.H. "Real Time Programming and Motor Control - ME 230 Class Notes." Unpublished-,1985.

Barnes, J.P.G., An Overview of Ada. Software-Practice and Experience, V10, pp.851-887, 1980.

Barnes, J.P.G., Programming in Ada, Addison-Wesley PublishingCo., London, 1982.

Bergman, Bruce, et. al. "Ada Compilers: Mission - Critical Software for the PC." in Computer Language, 3:12 and 4:1, December 1986, January 1987, Parts I & II.

Booch, Grady, Software Engineering with Ada, Benjamin/Cummings Publishing Co., Menlo Park, CA., 1983.

Bradley, Stephen P., Applied Mathematical Programming, Addison-Wesley Publishing Co., Reading, MA., 1977.

Burns, Alan, Concurrent Programming in Ada. Cambridge University Press, Cambridge, 1985.

Condiz, Marin. "Packages: Ada's Integrated Circuit," in Computer Language, 3:12, December 1986.

Craig, John J. Introduction to Robotics: Mechanics & Control, Addison-Wesley Publishing Co. Menlo Park, CA. 1986.

Gehani, Narain, Ada, An Advanced Introduction, Prentice-Hall Inc.,Englewood Cliff, N.J., 1983.

Goodwin, Graham, and Sin. Adaptive Filtering Prediction and Control. Prentice-Hall,Inc., Englewood-Cliffs, N.J. 1984.

Kim, Leonard. "Influence Diagram Based Expert System Tool Developed in Ada." unpublished, 1987.

Landau, Yoan D. Adaptive Control: The Model Reference Approach.Marcel Dekker, Inc., New York, 1979.

Norton, Peter, Programmer's Guide to the IBM PC, Microsoft Press, Bellevue, WA., 1985.

Reference Manual for the Ada Programming Language, U.S. Department of Defense, January 1983.

Sastry, Shankar and Bodson, Marc. "Stability, Convergence, and Robustness of Adaptive Systems - EE 290B Class Notes." to be published with Prentice-Hall, Englewood-Cliffs, N.J. 1986.

Stein, Matthew and Kim, Leonard. "Implementation Survey in Adaptive Control." EE 290B Final Project Report, December 1986.

Tomizuka, Masyoshi. "Advanced Control Systems - ME 232 Class Notes." Unpublished 1986.

1LT Leonard S. Kim has a BS in Operations Research from Princeton University, a MS in Mechanical Engineering from U.C. Berkeley, and a M.Eng. in the area of control theory also from U.C. Berkeley. In 1985, he was selected to participate in the Army's Technological Enrichment Program. Under the auspices of that program, LT Kim studied robotics and Ada as a graduate student at Berkeley for two years.

LT Kim is presently en route to Korea for a one year tour of duty with the 2nd Infantry Division.

A GRAPHICS ORIENTED DESIGN METHODOLOGY FOR REAL TIME CONTROL SYSTEMS USING ADA

Geoffrey C. Hingle
Steven L. Gilbert
Murat M. Tanik

Southern Methodist University
Dallas, TX 75275

Abstract

Current graphical system design methodologies do not support documentation of real-time system level design decisions such as the rate of data computation and the rate of inter-subsystem transmissions. We suggest extensions to a popular existing graphics notation for the Ada programming language (Buhr diagrams [1]), and demonstrate our notation's ability to clearly document data calculation and transmission rates. In addition, we extend the Buhr diagrams to include an Ada data dictionary that documents the structure of the interface between two objects. The format of the data dictionary lends itself to subsystem interface definition and coordination, which in large systems can involve more than one corporation. Two examples are included, the first models a cruise control system using an original Buhr diagram, and the second models a hypothetical radar detection system using our extended notation.

Key Words

Ada, Object-Oriented Design, Computer Aided System Design, Knowledge Based Systems

Introduction

As software based systems grow in size and complexity, software quality and trustworthiness problems become increasingly difficult to manage. Overruns in cost and schedule by factors of two, three, or more over original estimates are common. Serious problems can arise after the delivery of a product which are undetected during development. When changes to the software are required, further problems arise as features unrelated to the ones changed are inadvertently affected. Based on industry estimates, DoD mission-critical software costs will exceed $32 billion annually by 1990, constituting more than 10% of the annual defense budget. The current DoD trend is toward mission-critical systems that require large amounts of increasingly complex software in order to meet their performance criteria. This is evident in lengthening software development lead times (from five months in 1965 to 100 months for a typical application program today) [2].

There is a basic flaw in the idea that once a large software based system is delivered to the customer, the software in that system is an "off the shelf part" that does not have further life cycle costs. One is inclined to think that once the software is built, it will always perform the same function; therefore, the only costs for a piece of software are its development costs. It is true that for a particular piece of software there are no recurring costs for maintaining the physical software package whereas hardware needs to have components replaced when they fail. However, a large software based system goes through many stages of maturity and its software has a usefulness life time. Just like a battery in a hardware system, when the system software no longer meets the needs of the system (changing customer requirements), it too like a discharged battery must be replaced or recharged.

Impressive statistics have been gathered on the increasing costs for fixing (recharging) software oriented system problems throughout the system's life time (period of usefulness) [3,4]. This is the classical maintenance problem and it is many times more expensive to correct a problem after the system is delivered. Accordingly, the thrust of today's structured design methodologies as a whole is the front end of the project (i. e., putting as much work as possible into getting the requirements established before starting the design), thus taking advantage of the tremendous cost leverage of finding problems early. However, this approach does not protect against changing customer requirements [5].

The software engineer and system designer need system-oriented design tools that can be used in the original development as well as during the maturing phases of a system's life time. These tools must be relatively standardized because often the people that did the original design of the system are not the people who have to upgrade/maintain these large software based systems [6]. Formalized system design methods that lower the overall development cost of software based systems would serve to lessen the impact of changing customer requirements. These design methods should provide a structure for the representation of system level design requirements and coding techniques such that it is relatively easy to detect errors. In addition, a formalized design methodology would make changes to a software system late in its life cycle easier and reduce the risk of disrupting the system operation.

In addition to the lack of system-oriented design tools, there is also a shortfall of between 50,000 and 100,000 software professionals today. According to one estimate, if nothing is done to improve the current 4% annual incremental increase in software productivity, by 1990 the shortfall will be nearly one million [2].

The severity of the software productivity problem has become so obvious that twelve of the largest defense contractors in the United States have formed the Software Productivity Consortium as a centralized research facility to develop aerospace software technology. The Consortium became a corporation and limited partnership in September 1985. The companies that form the Consortium are: Allied-Signal (Bendix Aerospace), Boeing, Ford Aerospace, General Dynamics, Grumman, Harris, Lockheed, Martin Marietta, McDonnel Douglass, Northrop, Science Applications International, TRW, United Technologies, and Vitro.

The Consortium was established to create a basis for a dramatic increase in software productivity. The products produced by the consortium will exploit the following key concepts:

- Prototyping - The software prototyping tools and techniques will allow the software engineer to understand the needs of the customer and system. The software engineer will then develop solutions to these needs with the speed and precision offered by advanced automation.

- Reusable Software - Reusable software involves the construction of a system from pre-existing software components.

- Knowledge Base Systems - Knowledge-based systems capture expertise in a series of rules. The Consortium will use this technique to provide advanced automated assistance in project management, requirements definition, and software development.

- Software for System Engineering - The Consortium will also build information processing system libraries. Tools will be used to build requirements and design specifications, develop code, synthesize prototypes, perform dynamic assessment, and manage projects. When used as part of the integrated development environment, these tool sets will enable engineers to create and use libraries of reconfigurable software in the preparation of domain and application specific programs and systems. The information processing system libraries will be use to automatically generate requirement specifications and design code in specific interest areas, initially process control systems.

Considering this existing state of the art in developing large-scale software, we developed a set of requirements for a graphics and object oriented system design methodology.

Requirements for a Graphics and Object Oriented System Design Methodology

Carpenters use nails, wood, brick, steel, etc., to build houses. Mechanics rebuild motors using standard parts. Electrical engineers build microcomputers using standard chip sets. Mathematicians build proofs out of theorems, corollaries, and lemmas. These system architects all have one thing in common, that is, they build a system out of standardized parts using the tools of their trade. These standardized parts may have to be slightly modified using the tools of the trade for each application or design. But, the common thread in all of these approaches is the existence of a set of domain specific reusable components (i.e., nails, ICs, theorems), and tools for the manipulation and modification of these components. The domain-expert designer needs automated tools that support at least the following:

- The entire life-cycle for development and maintenance

- Communication among all levels of the design/programmer/test team members

- Guidance and help in problem analysis

- Top-down and bottom-up development

- Software, hardware, and system analysis/evaluation

- System evolution

Graphical system design notations which are intended to map directly to Ada constructs are discussed in literature [1,3]. System description techniques using these notations are loosely called system design methodologies, and are intended to be applicable for large real-time system design. It is important to note that in reality system design is far removed from the programming of a system, and hence the programming language. Design decisions made at the system level require hardware, software, budget, and time tradeoffs. In addition, the system designer must coordinate concurrent designs of subsystems with multiple contractors in an effort to construct a deterministic (testable) system. Therefore, when considering a graphical system design notation it is unreasonable to restrict the notation to only constructs that are directly relatable to programming language constructs, unless the program language is a true distributed system design language. For example, time-slicing design is commonly employed in large real-time systems during which data transfers (rendezvous) must take place deterministically [4,7]. This type of requirement specification is not considered by current graphical design notations. In order to develop extensions to the graphical system design notations suggested in [1] and [3], one needs a basic understanding of the constraints imposed by real-time systems.

The real world is very dynamic, and therefore quite different from the environment in which computers evolved. In addition, real time systems generally have to perform several tasks at once [8]. The cruise control system in an automobile is a common example of a real-time system which can illustrate the problems associated with real-time system development. Table 1 defines the I/O requirements of a cruise control system, and Figure 1 shows a data flow diagram of the system [9]. The Buhr diagram of this system is shown in Figure 2. It should be noticed that on this Buhr diagram timing constraints and detailed data structure are not incorporated (this is not currently part of the notation suggested by Buhr).

A Design Methodology for Large Systems

From the Strategic Defense Initiative to applications as mundane as dishwasher timing logic, computers are used as control units for systems which interact with their environment. These systems are generally called real-time systems because they are required to interact with their environments at precise times [10]. There are, however, unique difficulties surrounding this new application of technology. Failure in sensors or actuators, devices which carry out physical tasks, must be predicted and accounted for in the program controlling these devices. Traditional design programming methods should be expanded and new methods must be developed to handle the greater complexity involved with real-time systems. Currently, most embedded systems are composed of separate subsystems (i.e., embedded computing machines) that are controlled by a sequential operational program. This sequential program outputs required data to a double buffering system through which the data is then transmitted to other subsystems at a specified transmission rate [7]. Therefore, data that is to be transmitted at a certain rate is usually calculated at least as fast as the transmission rate. With such a protocol, a deterministic (testable) system can be developed.

Considering the characteristics of Ada and the defense department's requirements for the use of Ada, the basic structure of these deterministic systems is likely to continue. That is, a double buffering protocol for data to be transmitted across subsystem boundaries will most likely be used in Ada systems. It is unlikely that direct task communication will occur because of the

TABLE 1 I/O Requirements for a Cruise Control System

INPUTS	
System on/off	When on cruise control system should maintain the speed of the car.
Engine on/off	One pulse per revolution
Wheel pulse	Cruise control can only function when the engine is on.
Accelerator	How far the accelerator has been pressed.
Brake	Revert to manual if brake is on.
Increase/Decrease	Increase or decrease speed as requested by driver
Resume	Revert from manual state to last maintained speed.
Clock	Timing pulse at a fixed interval
OUTPUT	
Throttle	Digital value for the fuel supply system.

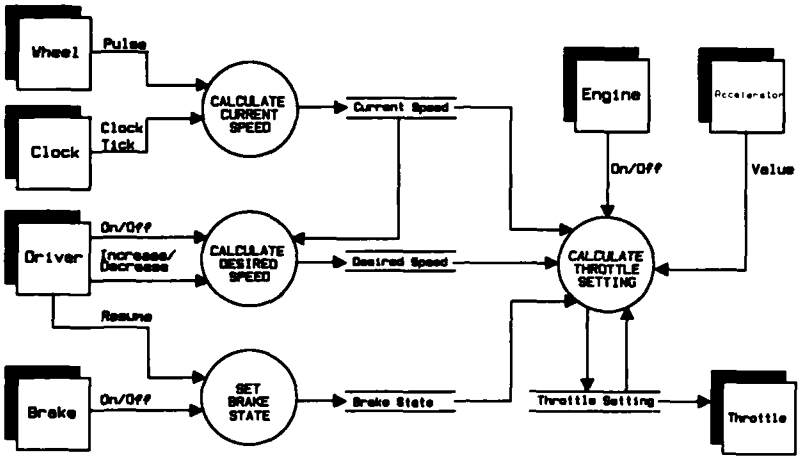


Figure 1 Cruise Control System Data Flow Diagram

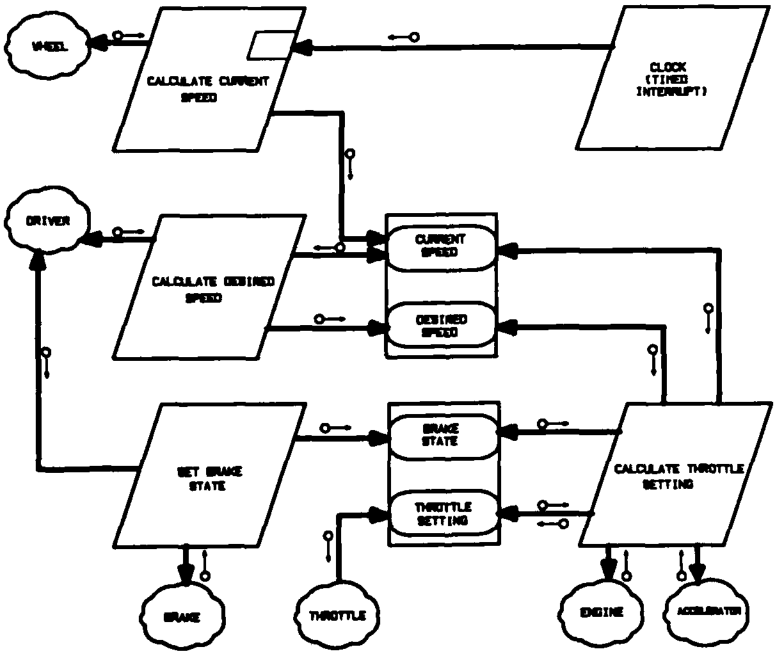


Figure 2 Cruise Control System Buhr Diagram

non-deterministic nature of the rendezvous, and the complications that arise when attempting to distribute a single program across several machines (which is required for an inter-subsystem rendezvous). However, even if sequential program units and double buffering are used when designing embedded Ada systems, each of the individual subsystems can be logically modeled as an Ada task when designing large distributed systems. If such a model is chosen, an existing graphic design approach (i.e., Buhr) can be utilized at the system design level when designing a system. This system model can then be directly passed to the next level of design, subsystem design, with all the abstractions made at the system level included in the model.

Although Buhr's notation for system design can readily be applied to the subsystem level of design, it cannot be applied to the distributed system design that requires data calculation timing requirements. In addition, knowledge of the structure and format of the interface is necessary. Therefore, we will introduce two extensions to Buhr's notation to provide this necessary system information:

- Mandatory Rendezvous Rate – this notation (see Figure 3) will be used to document the frequency at which a subsystem, logically modeled as an Ada task, must rendezvous at a defined task interface. In effect, this notation will be used to indicate the desired frequency at which the data transmitted in the logical rendezvous must be updated and stored in the transmission bus buffers. The requirements imposed by this notation cannot be directly mapped to an Ada construct, instead, these are system design requirements that must be met by the implementation of the entire subsystem operational program.

- Ada Data Dictionary – The data dictionary concept is used in methodologies that have been automated [11]. The Ada data dictionary (see Figure 4) is used to document the contents of the interface, and in our system, will also document the structure and format of the interface Ada types. In addition, the address of the data can be specified using an Ada address clause, and thus the data dictionary can be used as the configuration control document for the system interface.

To illustrate how these two extensions can be utilized to document system design level requirements, consider the following system that is used for detecting an enemy radar site and directing a missile to that site. It is partitioned as follows:

Navigation – One subsystem will be assigned the task of tracking the current location of the system. This subsystem will be responsible for supplying the system with current positional data in a fixed format (say latitude and longitude accurate to within a foot). Since most military applications require very fast and accurate positional information, this subsystem is required to transmit the current position 100 times per second (100 Hz). This unit will not require any input data from the rest of the system.

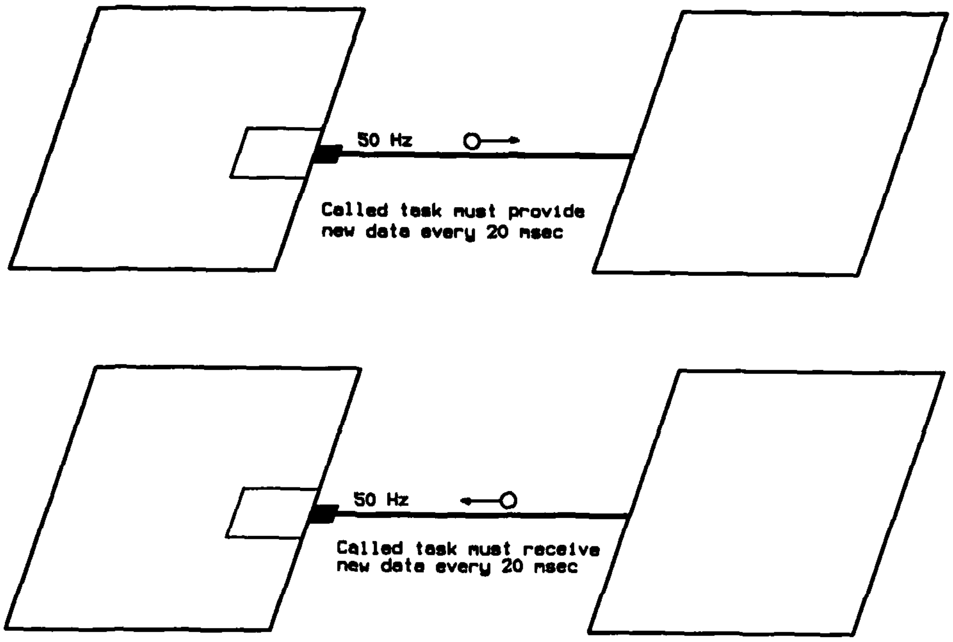
Detection – One subsystem will be assigned the task of detecting the enemy radar site. This subsystem will be equipped with a sensor tunable in the enemy radar frequency spectrum. This subsystem will receive positional data from the navigation unit, and if a site is detected, use this data to determine the location (latitude and longitude) of the enemy radar. This data is required at 50 Hz. In addition, this unit will receive control data (for frequency tuning) from the system controller at 25 Hz.

System Control – One subsystem will be assigned the task of system control, coordination, and control of the human user interfaces. This subsystem will detect subsystem failures, handle inter-subsystem communication, process human operator requests to change system parameters (e.g., the frequency of the radar detector), and drive the system display. Display control data from this subsystem will be transmitted at 50 Hz. Detector control data will be transmitted at 25 Hz. User request data from the display unit will be received and processed at 25 Hz.

Display Control – One subsystem will drive a video display which provides the operator a window to the system. It will receive data from the detector, the system controller, the weapon controller, and the navigation unit to construct a dynamic display necessary to provide the operator with sufficient situational awareness. In addition, the operator will interface with the system via this display.

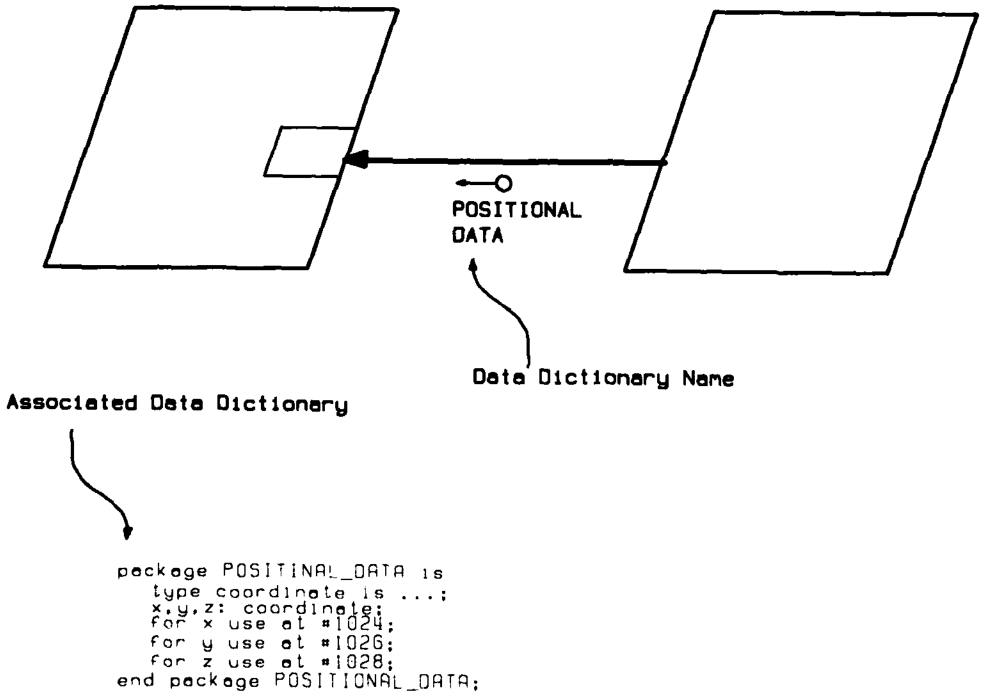
Weapon Control – Finally, one subsystem will be assigned the task of controlling any weapons that will be used to destroy detected radar sites. Data received from the system controller or directly form the navigation and detection subsystems will be used to aim the weapons correctly. In addition, this unit will deliver the ordinance when proper conditions (including operator consent) are met. Data transmitted to the weapon will be at 25 Hz also.

In this relatively complex example, the following design decisions are necessary:



NOTE: Caller task cannot force a rendevous to occur faster than than specified rate.

Figure 3 Mandatory Rendezvous Rate



```
package POSITINAL_DATA is
   type coordinate is ...;
   x,y,z: coordinate;
   for x use at #1024;
   for y use at #1026;
   for z use at #1028;
end package POSITIONAL_DATA;
```

Figure 4 Ada Data Dictionary

- System Partitioning - Which functions are to be performed by each subsystem so that collectively, the system requirements are satisfied.

- Subsystem Interface - What data is necessary from each subsystem in order that other subsystems may perform their functions

- Data Transmission Rates - Update and transmission rates for each subsystem's data such that the response time of the system to the real-world is adequate.

With the two simple additions to Buhr's notation, this extended notation can be used to implement an automated graphics oriented system design methodology to design and document our example radar site detection system. All of the timing requirements are documented in our model by applying mandatory rendezvous rate requirements to the data transfers that occur between each subsystem (see Figure 5). Ada data dictionaries defining the data transmitted among the different subsystems can be developed and documented as part of this diagram when the structure of the interface is determined.

Approach for Automation – An Expert System

We are currently investigating an implementation approach for the graphical design methodology discussed above. We believe the recent developments in knowledge based expert systems can be utilized to implement an intelligent interface to a software engineering database to provide a powerful system design tool [12]. This knowledge based system will provide configuration management and support communication between the different working groups that support the software life cycle. Our emphasis is on the use of the object-oriented paradigm for system development [13]. We are evaluating several programming languages, namely C++ [14], Objective C [13], and Ada [3, 15], as possible implementation languages. In addition, we are examining the functional ramifications of our assumptions through the use of exploratory prototypes on Sun stations and TI-Explorers using ART and other tools.

Acknowledgments

We would like to acknowledge valuable discussions on the nature of automated graphics systems with Raymond Yeh. Also, we would like to express our gratitude to the following colleagues of ours for their insights into the current state of software engineering: D. E. Sundstrom, Manager Systems Engineering; S. A. Alford, Staff Specialist Software Systems; and F. L. Corley, Engineering Lead System Design.

References

1. R. J. A. Buhr, "System Design with Ada," Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.
2. J. F. J, "Aerospace Software Consortium Established," *Defense Electronics*, Industry News, December 1985.
3. G. Booch, *Software Engineering with Ada*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, May 1983.
4. Duncan A. Mellichamp, "The Structure of Real Time Systems," *Real–Time Computing*, Von Nastrand Reinhold Company, New York, NY, 1983.
5. B. W. Boehm, "Software Life Cycle Factors," *Handbook of Software Engineering*, Von Nostrand Reinhold Company, New York, NY, 1984.
6. M. P. Mariani, D. F. Palmer, "Software Development for Distributed Computing Systems," *Handbook of Software Engineering*, Von Nostrand Reinhold Company, New York, NY, 1984.
7. *Aircraft Internal Time Division Command/Response Multiplex Data Bus*, MIL-STD-1553B, United States Department of Defense, February 1980.
8. D. L. Parnas, "Software Aspects of Strategic Defense Systems," *Comm. ACM*, Vol 28, No. 2, Dec 1985.
9. G. Booch, "Object-Oriented Development," *IEEE Trans. Software Engineering*, Vol SE-12, No. 2 Feb. 1986.
10. P. D. Lawrence and K. Mauch, *Real–Time Microcomputer System Design: An Introduction*, McGraw Hill, New York, NY, 1987.
11. W. Yin, T. J. Lee, M. Tanik, D. Y. Y. Yun, "Software Reusability and Knowledge Engineering: A Reusability Experiment," Working Paper, SMU, 1987.
12. R. Keller, *Expert System Technology*, Yourdon Press, Englewood Cliffs, NJ, 1987.
13. B. J. Cox, *Object Oriented Programming: An Evolutionary Approach*, Addison-Wesley Publishing Company, Reading, MA, 1986.
14. B. Stoustrup, *The C++ Programming Language*, Addison-Wesley Publishing Company, Reading, MA, 1986.
15. *Reference Manual for the Ada Programming Language*, United States Department of Defense, Ada Joint Program Office, July 1982.

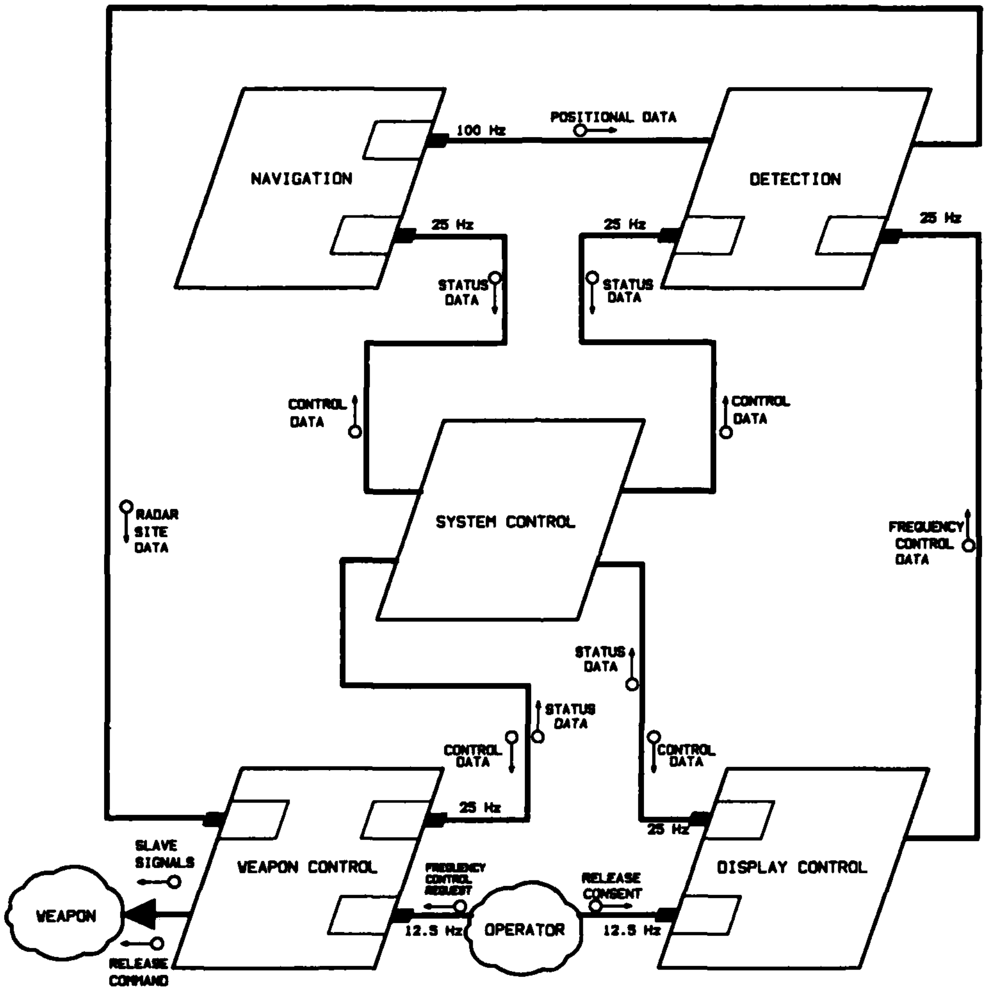


Figure 5 Radar Site Detection System

RACE CONTROL FOR THE VALIDATION AND VERIFICATION OF ADA MULTITASKING PROGRAMS

by

Tzilla Elrad :: Illinois Institute of Technology

Fred Maymir-Ducharme :: A T & T Bell Laboratories

ABSTRACT:

Parallel processing has increased the efficiency of concurrent programming and the complexity of software analysis; but these benefits have cost us the luxury of reliably testing and debugging programs that predictably behaved deterministically and sequentially. The repeated execution of a sequential program using the same input will invariably traverse the same legs of code in the program and therefore consistently produce the same results every time. Whereas, a concurrent program may be repeatedly run with the same input and never traverse the same legs of code, producing different results. The additional complexity is due to the introduction of nondeterminism and parallelism. *Controlling the nondeterminism in concurrent programs* is essential in order to test and debug Ada programs.

Testing and debugging real-time systems in Ada, using supplementary tools, can be very difficult and tedious. This process can be greatly simplified and facilitated by the enhancement of the Ada language feature set to support testing and debugging. The *preference control construct*,a proposed Ada feature, can be utilized as a tool for the testing and debugging of Ada programs. Using preference control as a testing tool to control the race amongst alternatives within a task has several advantages over some of the previously suggested testing and debugging tools. Preference control will have very little interference with the normal environment during testing, unlike separate testing processes, clocks or monitors that must be added to the environment and may cause bottle necks and unnecessary timing anomalies. Our solution insures that the semantics of the program are preserved and that the environment is not disruptively altered for testing. Various situations exist that require that the testing not interfere with additional timing constraints. The preference control construct requires very little time overhead as a language construct. The possible automation of testing and debugging techniques in Ada using this construct is described and analyzed.

INDEX TERMS : distributed computing, parallel processing, Ada, multitasking, concurrent programming languages, nondeterminism, preference control, race control, availability control, testing and debugging tools.

1. INTRODUCTION:

Distributed computing environments have led to the need for nondeterministic language constructs to exploit the available parallelism. The need then arose to control nondeterminism to increase expressive programming power. In [7,8] we classified these controls and introduced the preference control construct to further control nondeterminism in Ada. Parallel processing can result in programs that nondeterministically follow many different computation paths. Tools are needed to thoroughly test and debug concurrent Ada programs. These tools must allow users the ability to reproduce specific computation paths for debugging purposes and the reverification of results associated with the specific set of computation paths. Nondeterminism makes it virtually impossible to efficiently tests all of the possible computation paths without altering the semantics of the program; in fact, without a tool to control nondeterminism, one cannot be assured that all possible execution paths will be computed, regardless of the number of test runs.

Various schemes have been proposed to deal with this unpredictable program behavior. One solution entails controlling the scheduler via interrupts, delays, process priority and ordering processes in the ready queue [4,17] is very complex and requires an intimate knowledge of the operating system and its internals. The timing overhead involved with the "alarmclock monitor" or process [19], another proposed solution, limits its application to virtual-time environments; this is clearly unacceptable for real-time applications which must consider real-time constraints which are part of the decision making process. Another strategy consists of controlling the execution sequence of entry calls using a monitor as a buffer [5], this removes all nondeterminacy, since all events with a monitor are deterministic; this scheme, therefore, cannot be applied to a select statement with more than one accept alternative. Reproducing rendezvous sequences by only allowing each entry call to be issued after the previous call has been accepted [20] also uses a an additional task as a buffer; this solution will not cover the cases when the program requires two or more arrivals of entry calls between two consecutive rendezvous.

Using preference control as a testing tool to control the race amongst alternatives within a task has several advantages over some of the previously suggested testing and debugging tools [16,22,23]. Race control does not bear a significant timing overhead. Preference control does not have any effects on the runtime execution of tasks and the additional time overhead is insignificant when compared to the overhead associated with the addition of a monitor or separate synchronization process; this is necessary to avoid the masking of possible synchronization problems. Race control does not restrict existing synchronization by ignoring other entry calls, or not allowing them during testing. There are two basic approaches to controlling nondeterminism during testing. One strategy is to control the availability of each entry (hereafter referred to as availability control) , and the other strategy is to control the race between different entries within the same nondeterministic construct (henceforth referred to as race control.) Preference control is used to manipulate the race control.

We begin with a summary of the classification of nondeterminism controls and the preference control construct. The uses of preference control construct as a testing and debugging tool to manipulate race control and availability control for a distributed environment are then described and compared to other suggested tools. A Concurrent Readers & Writers example is used to illustrate how preference control can be utilized. The automation of testing and debugging using the preference control construct is later covered.

2. MANIPULATING RACE AND AVAILABILITY CONTROL FOR TESTING PURPOSES

Pure Nondeterminism: an unconstrained choice from a finite number of alternatives. Pure nondeterministic constructs are not sufficiently structured for today's concurrent programming language needs. The classification of controls on nondeterminism facilitates the learning and understanding of nondeterministic constructs. A better understanding of these controls allows the programmer to become more efficient in controlling nondeterminism and thereby exploiting the available parallelism to a larger degrees. The classifications of controls on nondeterminism also aids in testing and debugging by allowing the tester to explicitly specify the different execution paths desired, using the different controls.

2.1 AVAILABILITY CONTROLS ON THE SELECTION PROCESS

Hoare uses the notion of ***guards*** to denote the availability of an alternative for selection. If all of the guards for each alternative hold true, then one alternative is selected nondeterministically; otherwise, only the alternatives with true guards are considered for selection. We will classify this selection criteria amongst alternatives, controlled by the state of the alternative's guard, as **Availability Control**. Availability Control is the mechanism used to ***enable*** or ***dissable*** each alternative's guard, thereby controlling the domain of alternatives available for selection. Availability Control can be further sub-divided into the following categories:

2.1.1 PRIVATE CONTROL

Private control: Nondeterminism restricted by conditions over variables local to the task. Private control is considered open if the condition is true; otherwise it is closed. Private control was not intended to be allowed in Ada (but it can be implemented in Ada using a combination of the "when" and the "delay" primitives within the select statement.) When private control is specified for an alternative, it can only be chosen if it is open. Private control restricts the nondeterministic choice by not considering alternatives with closed private control.

2.1.2 CONSENSUS CONTROL

Consensus control: nondeterminism restricted by environmental/communication constraints. The alternatives are restricted to only those for which the communication is available from the rest of the system. Consensus control is considered established if the rendezvous can be established. This control can be attained in Ada by the use of the "accept" primitive within the select statement.

An alternative is ready if all of the controls have been satisfied; that is, if the private control is open and/or if the consensus control is established. The *nondeterministic* construct will check all of the controls in each alternative and then choose one of the ready alternatives. In many cases, the use of the nondeterministic construct includes using a combination of these controls.

2.2 RACE CONTROL ON THE SELECTION PROCESS IN ADA

Beyond Availability Control, there exists other races between different groups of entities and at different levels. Priorities are managed and supported at the operating system level. Other controls are exercised at the programming language level. We will classify the following controls at different levels as follows:

1. PRIORITY CONTROL

   Controls the race amongst different tasks at the operating system level. In Ada, if two of more tasks with different priorities are in the ready state, the task with the highest priority will be selected for running.[3]

2. PREFERENCE CONTROL

   Controls the race amongst different alternatives within a task. We have classified preference control as nondeterminism restricted by a preferential order; preference control gives the programmer the power to assign preferences to the alternatives within the nondeterministic construct. Each alternative inside a nondeterministic construct may (but need not) have a preference value. A lower value indicates a lower degree of urgency; the range of preferences is implementation defined. (i.e. a ready entry with a preference = 2 is chosen before a ready entry of preference = 1). Preference control gives nondeterminism a defined relational order between alternatives.

```
pref <constant> when <condition> => accept <entry> ...
```

Figure 1. SUGGESTED SYNTAX OF THE PREFERENCE CONTROL CONSTRUCT IN ADA

```
select
  pref 3 when B1 => accept entry1(...)
      do S1; end entry1
or
  pref 2 when B2 => accept entry2(...)
      do S2; end entry2
or
  pref 2 when B3 => accept entry3(...)
      do S3; end entry3
or
  pref 1 when B4 => accept entry4(...)
      do S4; end entry4
end select;
```

Figure 2. THE PREFERENCE CONTROL CONSTRUCT IN A SELECT STATEMENT

All of the nondeterminism constraints are listed before the entry call: first, the preference control (pref constant); followed by the private control (when < condition > =>); and finally, the consensus control (accept <entry>). If a preference is not specified, it should default to the lowest value (ie. if negative values for preferences are not allowed, then default to 0). The same value preference can be assigned to different entries within the same select statement to allow a greater amount of nondeterminism.

There are some similar language constructs in existence. The "cell" concept [18] allows explicit preference control amongst the entries within the *select statement* by attaching labels to entries and ordering the labels statically. And Occum [17] has a construct called **PRI ALT**, (stands for priority alternatives) which implements implicit static preference control by allowing alternative processes to be prioritised in the order they are listed within the **PRI ALT** construct. Both Occum and Cell contain a construct for static preference control, which do not allow the assignment of the same preference to more than one alternative.[1] This paper specifically deals with static preference control to ensure reproduceability by not allowing the changing of the preferencial orderings during runtime. Dynamic preference control is more applicable and powerful in the programming arena [9,10] . There exist some situations in which dynamic preference control can benefit testing and debugging; these are out of the scope of this paper.

3 TESTING AND DEBUGGING IN ADA'S DISTRIBUTED ENVIRONMENT

Testing and debugging sequential programs is a complex issue. Concurrent programs naturally have similar difficulties; these difficulties are out of the scope of this paper. The intent of this paper is to address some of the specific issues which result from the added nondeterminism and parallelism to concurrent programs. These issues are unique to concurrent programming in a distributed environment.

The added complexity of testing concurrent programs is a result of the new semantics introduced by concurrent programs. Let **SEQ** denote a sequential deterministic program. The denotational semantics of **SEQ** is a function from an initial state and environment to a unique final state and environment.

$$M\,[[\ \mathbf{SEQ}\ ]]\ (\ Si,\ Ei\ ) = (\ Sf,\ Ef\ )$$

WHERE:

M is the symbol for the denotational semantics function.

(Si, Ei) is the initial state and environment.

(Sf, Ef) is the final state and environment.

With respect to testing and debugging **SEQ**, the denotational semantics imply that given a specific input, the execution of **SEQ** will always result with same output.

Let **DIS** denote a concurrent program. The denotational semantics of **DIS** is a function from an initial state and environment to a set of final states and environments.

$$M\,[[\ \mathbf{DIS}\ ]]\ (\ Si,\ Ei\ ) = \{\ Sx,\ Ex : x = 1 \ldots n\ \}$$

WHERE:

{ Sx, Ex : x = 1 ... n } is the set of possible final states and environments.

With respect to testing and debugging **DIS**, the denotational semantics imply that given a specific input for a concurrent program, the execution of **DIS** may have many different results. Parallel processing is asynchronous and therefore the order of actions can be different every time the program is executed. Another reason we can expect different results is that if nondeterministic constructs are used, a different set of alternatives can be nondeterministically chosen each time the program is executed. Multi-tasking can result in a nonpolynomial order of different results; therefore, testing cannot be complete until all possible results are tested. Because of the parallelism and asynchronicity of concurrent programming languages, it may be impossible to produce all possible results.

Another testing issue is that of reproducibility. If an error is found during a run, one must be able to reproduce the same error by running the exact same sequence of actions and interaction of tasks.

Debugging a parallel program is also more complex when alternatives are chosen nondeterministically. One may wish to debug one sequence of multi-task rendezvous and not be able to reproduce it consistently. If one is unable to "step" through the program sequentially and deterministically, debugging will be nearly impossible.

[1] Burns [6] suggests an enhanced select statement called priority select, which is static, but can allow the assignment of the same preference to several alternatives by nesting select statements within priority select statements

4. THE TESTING AND DEBUGGING TOOL FOR ADA PROJECTS

The repeated execution of a sequential program using the same input will invariably traverse the same computational path in the program and therefore consistently produce the same results every time. Whereas, in an ADA concurrent program, one may make repeated runs with the same input and never traverse the execution path, producing different results. The additional complexity is due to the introduction of nondeterminism. Controlling the nondeterminism in ADA tasking is essential in order to test and debug these programs. The preference construct can be utilized to control this race by assigning appropriate values to the alternatives involved in the desired computational paths. This will not necessarily "force" the desired path, but it will give it a much higher probability. The other factors that must be taken into account are environmental; they include availability controls and synchronization. For example, assume the desired path to be tested required that *alternative2* be selected from a select alternative with : alternative1, alternative2, alternative3 and alternative4. Giving *alternative2* the highest preference would only ensure that it would be chosen if its private control was open and its consensus control established; otherwise, another ready alternative would be chosen. Ignoring other alternatives and "forcing" alternative2 to be chosen via a separate monitor or task is not a suggested practice; it does not adhere to software engineering principles. Manipulating race control does not change the semantics of the program; it changes the probabilities to allow more efficient and controlled testing and debugging.

The preference control construct will also allow the manipulation of the availability control; therefore allowing the "forcing" the selection of specific alternatives. This is accomplished through the assignment of **pref 0** to the alternatives other than the desired alternative. The selection of alternative2 in the latter example could have been forced by assigning alternative1, alternative3 and alternative4 a value of "0" preference; this instructs the compiler not to consider those alternatives for selection. Pref 0 has a much wider and powerful application when used in dynamic preference control. [10]

4.1 A SIMPLE EXAMPLE TESTING AND DEBUGGING SCENARIO

Lets assume that we would like to test the following well known example of the Readers - Writers Solution, which is implemented in Ada [11].

```
task RW is
  entry START_READ;
  entry END_READ;
  entry START_WRITE;
  entry END_WRITE;
end RW;

task body RW is

 NO_READERS: NATURAL := 0;
 WRITER_PRESENT: BOOLEAN := FALSE;

begin
 loop
  select
   when not WRITER_PRESENT =>
       accept START_READ;
          NO_READERS := NO_READERS + 1;
   or
       accept END_READ;
          NO_READERS := NO_READERS - 1;
   or
   when not WRITER_PRESENT and
      NO_READERS = 0 =>
       accept START_WRITE;
        WRITER_PRESENT := TRUE;
   or
       accept END_WRITE;
         WRITER_PRESENT := FALSE;
  end select;
 end loop;
end RW;

package body RESOURCE is

 S: SHARED_DATA :=  -- the shared data.

 -- the specification of task RW comes here.
 -- the body of task RW comes here.

 procedure READ(X: out SHARED_DATA) is
  begin
   RW.START_READ;
   X := S;
   RW.END_READ;
  end READ;


 procedure WRITE(X: in SHARED_DATA) is
  begin
   RW.START_WRITE;
   S := X;
   RW.END_WRITE;
  end WRITE;

end RESOURCE;
```

Figure 3. R_W WITHOUT RACE CONTROL

Example 4 below has modified the body of task RW to illustrate the use of preference control as a testing mechanism.

```
task body RW_wPREF is

 NO_READERS: NATURAL := 0;
 WRITER_PRESENT: BOOLEAN := FALSE;

-- ADDING PREFERENCE CONTROL FOR TESTING

begin
 loop
  select
   pref -> a: when not WRITER_PRESENT =>
          accept START_READ;
             NO_READERS := NO_READERS + 1;
```

```
    or
    pref -> b:
            accept END_READ;
               NO_READERS := NO_READERS - 1;
    or
    pref -> c: when not WRITER_PRESENT and
            NO_READERS = 0 =>
            accept START_WRITE;
               WRITER_PRESENT := TRUE;
    or
            accept END_WRITE;
               WRITER_PRESENT := FALSE;
  end select;
 end loop;
end RW;
```

Figure 4. R_W WITH RACE CONTROL

The body of RW in example 3 was modified in example 4 to include preference control. We assigned a preference of "a" to START_READ, a preference of "b" to END_READ, and a preference of "c" to START_WRITE. You will notice that no preference was assigned to END_WRITE; the justification of this decision is described in section 6.

4.2 RESULTS OF USING PREFERENCE AS A TESTING TOOL

The Readers - Writers Problem is very well known and has been studied for quite some time; therefore, the results of testing are also well known. We chose this example to demonstrate the power of using preference control as a testing mechanism to reach the same results.

There are two separate problems that arise in this example:

1. CASE 1 - Starvation of the writers can occur when the readers are continuously serviced. A set of runs with the assignment of : "a = 3, b = 2, c = 1" will greatly increase the likelyhood of starving the writers by giving a higher preference to the serving of readers.
2. CASE 2 - Starvation of the readers can occur when the writers are continuously serviced. A set of runs with the assignment of : "a = 1, b = 2, c = 3" will greatly increase the likelyhood of starving the readers by giving a higher preference to the serving of writers.

As previously mentioned, the use of preference control will not guarantee the choosing of one alternative over the other; this is due to the nature of nondeterminism and the use of the other controls of nondeterminism. In this case, preference control was used to test different scenarios to surface problems that may have taken many more runs to find without the use of preference control.

5 AUTOMATING THE TESTING PROCESS

A preprocessor can be implemented to automate the testing process by adding preference control to the program that is to be tested. The preprocessor should follow the following steps for each of the nondeterministic constructs included in the program:

1. STEP 1: Assign a preference control to each alternative within the nondeterministic construct.
2. STEP 2: For constructs with **k** alternatives, the preprocessor will output **k**! permutations of different preferences for the different alternatives. This should not be an unacceptable number of permutations, since **k** is in most cases of order three or four.
3. STEP 3: Assign preferences to all of the necessary alternatives of each task one at a time, leaving the other tasks unchanged. This will allow better control of the flow of the program.
4. STEP 4: Assign preferences to all the pertinent alternatives in each task simultaneously. This may cause an explosion of all the possible cases. (ie. if we have **ki** alternatives in task **ti** for i = 1 ... 5, we will have **k1! * k2! * k3! * k4! * k5!** different sets to test.)

OPTIMIZATION TECHNIQUES:

Consider task RW in the Readers - Writers problem. The semantics of the task implies that RW will never be in a state in which the END_WRITE alternative is ready and any of the other three alternatives is also ready. In other words, END_WRITE never competes with the other alternatives within the nondeterministic construct; hence, the use of preference control on the END_WRITE alternative would not have any impact on the nondeterministic decision. In general, when the semantics of a nondeterministic construct denotes that two or more alternatives are never ready simultaneously and therefore never have to compete with each other, there is no need to assign preference control to those alternatives.

6 SEMANTIC ISSUES

One of the major difficulties of testing concurrent software is that the testing procedure may itself change the semantics of the software being tested. The results then, may not be valid since a new variable was introduced into the execution of the program. Adding a special t processor as a tester or a monitor must be very ca designed so as not to interfere with the original synchronization. The addition of preference control increase the control on the nondeterministic choice alternatives does not affect the denotational semantics of concurrent programs.

Let **DIS** be a concurrent program and let **EXTENDED-DIS** be its extension with the assignment of preference control for testing.

$$M [[\ \mathbf{DIS}\ ]] (\ Si, Ei\ ) =$$

$$M [[\ \mathbf{EXTENDED\_DIS}\ ]] (\ Si, Ei\ ) =$$

$$\{\ Sx, Ex : x = 1 \dots n\ \}$$

The effect of adding preference control to a program increases the probability of a specific result. Dynamically changing the preference control will allow the programmer to systematically increase the probability of deriving each of the possible results. Moreover, when an error is detected in the output of a run with a specific set of defined preferences, the tester can use preference control for debugging purposes. In this case, the tester would simply use the same set of preferences to increase the probability of traversing the same computational paths that produced the error. This would also allow the tester to better understand and trace the computational paths executed during debugging.

Executing a set of tests with the same preferences is very likely to reproduce the same error during each run. This can be a very powerful tool for testing and debugging. In a distributed environment, preference control is simple, free of side effects and a very powerful tool for the testing and debugging of parallel programs.

7. CONCLUSION:

Nondeterminism is a central concept and issue in modern concurrent programming languages that exploits explicit parallelism. Controlling nondeterministism and parallelism without changing the semantics or the environment of the program is essential for the validation and verification of multitasking ADA programs.

Simple and efficient testing and debugging tools for today's distributed environments are essential. The added complexity and difficulty in testing and debugging, introduced by concurrent programming languages, requires *new testing tools and methodologies*. *We suggest the* addition of preference control to the Ada language feature set to allow race control as a testing and debugging tool for distributed environments. Compared to other testing and debugging tools, race control appears to be the simplest and most powerful choice for testing nondeterminism. Unlike other solutions, preference control does not alter the environment by introducing a separate process for testing and it does not change the semantics of the program. There is no significant time overhead associated with race control. And race control does not restrict the system synchronization or the state of other conditions (ie. availability controls.) We have implemented preference control in Concurrent C, which is very similar to Ada: this will be covered in our next paper. An ADA pre-processor for adding preferences is also available.

REFERENCES

[1] G.R.Andrews, F.B.Schneider, "Concepts and Notations for Concurrent Programming" 1983 ACM 0010-4892/83/0300-0003.

[2] G.R.Andrews, Synchronizing Resources," ACM Trans. Programming Languages Syst. Vol. 3, Oct 1981.

[3] G.Booch, "Software Engineering with Ada" The Benjamin/ Cummings Publishing Co.,Inc 1983.

[4] Brinch Hansen, P., "Testing a Multiprogramming System," Software-Practice and experience, Vol.3, 1973, 145-150.

[5] Brinch Hansen, P., "Reproducible Testing of Monitors," Software-Practice and Experience, Vol.8, 1978, 721-729.

[6] A. Burns, "Using Large Families for Handling Priority Requests," Ada LETTERS January, February 1987 Vol. VII, No. 1.

[7] T. Elrad, F. Maymir-Ducharme, "Distributed Language Design: Constructs for Controlling Preferences" Proceedings of the 1986 International Conference on Parallel Processing, August 19 - 22, 1986.

[8] T. Elrad, F. Maymir-Ducharme "Introducing the Preference Control Primitive: Experience with Controlling Nondeterminism in Ada", Proceedings of the 1986 Washington Ada Symposium.

[9] T. Elrad, F. Maymir-Ducharme "Efficiently Controlling Communication in Ada Using Preference Control" Proceedings of the 1986 IEEE Military Communications Conference, October 5 - 9, 1986.

[10] T. Elrad, F. Maymir-Ducharme "*Preference Control:* A Language Feature for AIDA Applications," Proceedings of the 1987 Third Annual Conference on Artificial Intelligence & Ada, George Mason University, VA, October 14 - 15, 1987.

[11] T. Elrad and N. Francez, "A Weakest Precondition Semantics for Communicating Processes", Lecture Notes in Computer Science, 13, 7 Springer-Verlag, 1982.

[12] T. Elrad, "A Practical Software Development for Dynamic Testing of Distributed Programs", Proceedings of the 1984 International Conference on Parallel Processing, August 1984.

[13] T. Elrad, "Data Dependencies Within Distributed Programs", *Proceedings of the Hawaii International* Conference on System Sciences, January 2, 1985.

[14] N. Gehani, "Ada: Concurrent Programming", Prentice Hall, 1984.

[15] N. Gehani, W. Roome "Concurrent C*" AT&T Bell Laboratories, Murray Hill, New Jersey 07974, 1985.

[16] D.Hembold, D.Luckham, "Debugging Ada Tasking Program", IEEE Software, March 1985 (Vol.2, #2)

[17] D.Hembold, D.Luckham, S.German "Monitoring for Deadlocks in Ada Tasking" Comm. ACM vol.7, 1982.

[18] C.A.R. Hoare, "Communicating Sequential Processes" Prentice Hall, 1985.

[19] C.A.R. Hoare, "Monitors: an Operating System Structuring Concept, Comm. ACM, Vol. 17, No. 10, Oct. 1974

[20] E. Horowitz, "Fundamentals of Programming Languages" Computer Science Press, 1984.

[21] J.D. Icbiah, et al, Reference Manual for the Ada Programming Language January 1983.

[22] Kuo-Chung Tai, "On Testing Concurrent Programs", IEEE Trans. Soft. Eng. 1985, Vol. p. 310 - 317.

[23] Kuo-Chung Tai, "An Approach to Testing Concurrent Ada Programs" Proceedings of the 1986 Washington Ada Symposium.

[24] B.Liskov, "Primitives for Distributed Programming" Computation Structures Group Memo 175 MIT Laboratory for Computer Science (May 1979).

[25] D. Luckham, F. vonHenke, "An Overview of ANNA, a Specification Language for Ada", IEEE Software, March 1985 (Vol.2, #2)

[26] Occam Programming System, VAX/VMX Host Manual, INMOS, 1983.

[27] S.A.Smolka. P.Wegner, "Processes, Tasks and Monitors: A Comparative Study of Concurrent Programming Primitives" IEEE Transactions Software Engineering 1983.

[28] P.D.Stotts, "A Comparative Survey of Concurrent Programming Languages" ACM SIGPLAN Notices, vol. 17, no. 9, September 1982.

[29] A.Silberschatz, "Cell: A Distributed Computing Modularization Concept," IEEE Trans. Softw. Eng., Vol. SE-10, No.2, March 1984.

[30] R. Taylor, "A General Purpose Algorithm or Analyzing Concurrent Programs", CACM, May 1983. (Vol.26, #5)

Fred A. Maymir-Ducharme received the B.S. degree in Computer Science from the University of Southern California in 1984, and the Ph.D. in Computer Science from the Illinois Institute of Technology in 1987, specializing in concurrent programming languages.

He is currently working on research and development at A T & T Bell Laboratories, in Naperville, Illinois. He is presently teaching a graduate class, "Software Engineering with Ada," at IIT. His areas of interest include multi-computers, parallel programming and distributed processing.

Tsilla Elrad, professor of Computer Science at the Illinois Institute of Technology, received the B.S. degree in Computer Science from the Hebrew University, the M.S. from Syracuse University in New York, and the Ph.D. in Computer Science from the Technion in 1982.

Tsilla's interests are in concurrent programmming language design, concurrent programming applications, testing and formal verification.

A Monoprogramming Approach to Host-based Systems

Andrea Di Maio

TXT S.p.A., Milano (Italy)

Abstract

Two different development approaches for soft real-time non-embedded Ada applications are compared: the multiprogramming and the monoprogramming ones. A particular monoprogramming design method is presented, which is based upon the Virtual Node concept and describes the system as a single Ada program. The method provides high flexibility thanks to separation of design and configuration phases, and it is viable for distributed systems. The approach and the support tools are briefly outlined, and a factory automation application is presented.

1. Introduction

Since the very beginning, Ada was conceived for embedded real-time applications, typical of military environments. The kind of applications Ada designers had in mind involved massive software developments and hard real-time constraints, needing both sound software engineering methodologies and run-time efficiency and reliability.

Now many potential Ada users are a bit worried by the performances of current Ada run-time systems and the language support to real-time itself. Ada provides a particular concurrency model based upon tasking, with fixed scheduling policies which do seriously affect real-time constraints handling. Recently such issues have been brought to light in order to suggest to address them during future language reviews [WRTAI 87].

Nevertheless there is a wide variety of applications which can be cost-effectively developed in Ada. The main features of such applications are:

- Soft real-time constraints : they are real-time applications, but the most time constants are relatively high.
- Parallelism : the most natural way to model them is by using concurrency;
- Size : hundreds of thousands source code lines, large development teams.
- Host-based : such applications can be regarded as embedded, but the target machines are more powerful than simple microprocessor-based boards. Generally they provide mass storage (disks, tapes), a general-purpose operating system, and some software tools. In other words they are much more similar to hosts than to embedded systems.
- Distribution : such systems are often distributed for both reliability and application-specific purposes (e.g. a plant control system performing both special device handling and statistical report generation could use different machines to monitor special devices and to perform statistical computations and report generations). Even if the target system is not physically distributed, the life-spans of subsystems are logically different (both batch and on-line activities coexist).
- Potential for reuse : more than one application of the same "class" are likely to be developed; therefore major code reuse should be enforced.

Many interesting examples of such applications can be found in military and industrial environments: command, control, communication and intelligence (C3I) systems, factory automation systems, power control systems. Since the target machine configurations used for these applications are quite complex and powerful (as said above), we will refer to them as host-based systems.

2. Development approaches

While a real-time embedded application can be designed using a monoprogramming approach, i.e. as a single concurrent program, using the underlying run-time support provided by the language, host-based systems are usually implemented as sets of communicating programs, each one mapped onto a single operating system process, which we will call Physical Node (PN). This approach is known as multiprogramming.

The disadvantages of this approach are evident.
First of all, interface checking between different subsystems enforced by high-level languages (and in particular by Ada) is lost if they communicate using operating system primitives and buffers.
The problem does not simply concern syntactical aspects, but especially the semantic of communications between subsystems. Typical operating systems provide a wide range of communication and synchronization primitives: synchronous, asynchronous, blocking, non-blocking, etc. This mainly requires a considerable initial effort in defining a subset of these primitives to be used within the project, whose output is an interface definition which strictly depends on the features of the basic available primitives. Moreover, some developers can easily overcome this interface, by using different OS primitives that are more convenient for their specific needs.

A major problem of the multiprogramming approach is low configuration flexibility. The allocation of (part of) subsystems to (Ada) programs (i.e. O.S. tasks) is made too early during the design process, whereas it would be useful to guarantee a certain degree of flexibility, in order to allow cheap reconfiguration of subsystems, due to load balancing, functional changes, etc.
Another consequence is decreased portability: if the system is to be ported on different machines, major parts have to be reimplemented.

This influences both technical and managerial aspects of the project: testing, quality assurance and configuration management are affected and the overall development times and costs incrase.

In order to avoid such drawbacks, the adoption of a monoprogramming approach for host-based applications should be enforced. At present Ada is the best candidate to provide all the needed features (support to concurrency, software engineering concepts, powerful environments) for monoprogramming developments. Nevertheless, the most Ada compilers for host-based systems map the whole Ada program onto a single O.S. process, which does not solve the problems mentioned above. Therefore both a design methodology and proper tools are needed to support monoprogramming for host-based systems.

3. Virtual Nodes

Previous research projects on distributed system programming in Ada [Tedd et al. 84, Dapra et al. 84] recognised that in order to develop a system which could be split into separate, possibly distributed, communicating components with the needed flexibility and efficiency, it is necessary to reflect the logical structure of the distributed target at the design level.
This is particularly true for the systems we have in mind, where, from the requirements specification phase, it is possible to identify different "functional nodes" with potentially different requirements in terms of computing resources, life-span, etc. In order to consider these disjoint nodes at the programming level, an abstraction of such nodes has to be defined, featuring high internal cohesion and low node coupling.
This abstraction has been defined Virtual Node (VN) [Tedd et al. 84, Goldsack et al. 87].
VNs are not to be confused with physical nodes (PN), that are target processors or operating system processes constituting the running system.

The basic idea is to allow full Ada within the boundaries of virtual nodes, and to constrain the communications between different virtual nodes to follow a single precise scheme.
This attitude results into both a structuring entity at the design stage and a quite efficient solution which reasonably bounds the overheads due to inter-node communication. The advantages are particularly evident for distributed systems, where communications between remote target machines are far less efficient than communications between Ada tasks running on the same machine, as they concern lower-speed communication means like serial lines. Nevertheless this is also true for non-distributed host-based applications of the kind described above: in this case there are intra-node communications within each PN belonging to the system, which are handled by the Ada run-time system, and inter-node communications which are performed using the primitives provided by the underlying operating system.

Starting from the VN abstraction, it is possible to define a design methodology using VNs as basic components. Since a VN is a functional abstraction, it has not to be immediately mapped onto a PN, but this configuration activity can be delayed until the design of all VNs has been completed. Obviously it is unrealistic to assume a complete independence of the VN design phase and of the configuration phase, as some configuration constraints will often influence the design phase: for instance a VN responsible for graphic screen management will run on a workstation and will have visibility of the bit-mapped screen as a device, while a disk manager VN is not likely to run on a diskless machine, and so on.

Nevertheless to enforce the separation of the two phases can meaningfully improve the whole development.

In order to meet the requirements of host-based systems VNs must possess certain fundamental properties, like complete encapsulation of their internal state (that can be only accessed through a proper interface), lack of reference to shared objects, own thread of control. It is interesting to note that many of these properties are required by the object-oriented design method, which therefore is a good candidate for VN-based developments.

One of the interesting features of a VN is that it can be completely defined in terms of Ada concepts. In the most general case it is the transitive closure of a procedure in an Ada library: the units of such closure must be compliant with a set of composition rules which provide the VN with the required properties.

The composition rules are thoroughly explained in [Goldsack et al. 87].

Communications between VNs are constrained to happen via remote entry calls.
The interface a VN provides to other VNs consist of an interface package containing a task which exports a set of entry calls. Such entry calls can be remotely called by other VNs if the interface package is visible to them ("with clause").

The whole application is a single Ada program made of many communicating VNs.
Nevertheless at the design stage the developer is unaware of which remote entry calls will rely upon the operating system inter-process mechanisms: all he has to consider are VNs.

Remote rendezvous can be handled at two different levels:

a) by the Ada RTS or

b) by the application program.

In the first case, the RTS should recognize whether a rendezvous is remote or not and invoke the proper operating system primitives. This would require an integration between the compiler and operating system greater than the most compilers provide. Furthermore, this solution is all but portable.

In the second case, parts of the application program must be changed at both sides of a remote rendezvous, in order to invoke the operating system primitives. Such changes would strictly depend on what the OS provides, and the advantages of the virtual node approach (configuration flexibility, portability) would be partially lost.

The only way to achieve the objectives of flexibility and portability (with respect to both compiler and operating system) requires:

1) the definition of a standard communication interface (SCI) to be implemented for different target operating systems;

2) the definition of a source-level transformation which transform each PN (i.e. a collection of VN after the configuration phase) into a different Ada program using SCI primitives;

3) the development of automatic tools supporting VN definition and transformation.

The first requirement consists of identifying which parts of an application program are directly influenced by the underlying operating system primitives. Surely the non-portable part of host-based applications are those concerning intern-PN communication. The SCI should be an Ada package with fixed interface and semantic, whose body would be re-implemented for each different operating system.

The second requirement aims to a multiprogramming system organization (i.e. one Ada program for each physical node) from the monoprogramming one.

The third requirement is a consequence of the previous two: if the source transformation is based on a fixed interface, it is likely to be automatable, which means that the final multiprogram solution is completely invisible to the developer and acts as a sort of low-level implementation of the concepts used at the design and configuration phase.

This approach to remote communication can be easily mapped onto the ISO/OSI layering scheme: source level transformations provide the presentation and session layers, the SCI provide the transport layer, while the rest is assumed to be available from the operating system.

4. The standard interface

The main requirements of the SCI are: support for dynamic creation of ports as communication end-points, use of system-independent names, asynchronous send and synchronous receive primitives, support for exceptions due to communication failures.

This requirements allows the SCI to be

a) independent of the underlying system

b) implementable on different systems

The SCI is presented in figure 1. It is worth noting that a quite elegant object-oriented and system-independent Ada representation of ports is given (a generic package instance is associated to each port) and that the private part contains system-dependent features (like the address structure enforced by the operating system).
Primitives for buffer allocation and deallocation are provided, together with non-blocking SEND and blocking RECEIVE. Their parameters are of generic types and are instantiated according to the parameters involved in each particular transaction.

The SCI package is relatively easy to port: it has been implemented first for Unix 4.2, using sockets as basic communication mechanisms, and it has been ported on VAX/VMS with approx. 1 man-day effort.

5. Tools

First of all, support tools are needed at the design stage. Since VNs are defined in Ada terms according to certain composition rules, it is necessary to check whether a given closure is a legal VN or not. Such checks can only be performed on correct Ada units and need a classification of such units which is not provided by the compiler. Therefore two tools, a classifier and a checker are needed [Moreton 87]. The first one classifies correct Ada sources along a set of properties (whether a unit declares objects, provides entries in its interface, etc.). Information collected by the classifier are then used by the checker that validates VNs.

The main role of the transformer [Moreton 87] is to process source code in order to create an Ada program for each PN.

```
package STANDARD_COMM_INTERFACE is
  type PN_ID is new NATURAL;
  type RETURN ADDRESS is private;
  type HEADER_TYPE is private;
  generic
    PN: PN_ID;
  package CALLER is
    generic
      type CALL_PACKET is private;
      type ANSWER_PACKET is private;
    package PRIMITIVES is
      type REF_CALL_PACKET is access CALL PACKET;
      type REF_ANSWER_PACKET is access ANSWER_PACKET;
      procedure ALLOCATE (A: out REF_CALL_PACKET);
      procedure DEALLOCATE (A: in out REF_ANSWER_PACKET);
      procedure SEND (A: in REF_CALL_PACKET);
      procedure RECEIVE (A: out REF_ANSWER_PACKET);
    end PRIMITIVES;
    procedure CLOSE;
  end CALLER;

  generic
    AN: AN ID;
    type REMOTE_ENTRIES is (<>);
    type CALL PACKET is private;
    type ANSWER_PACKET (R: REMOTE_ENTRIES) is private;
  package CALLEE is
    type REF_CALL PACKET is access CALL PACKET;
    type REF_ANSWER_PACKET is access ANSWER_PACKET;
    procedure ALLOCATE (A: out REF_ANSWER_PACKET;
                        R: in REMOTE_ENTRIES);
    procedure DEALLOCATE (A: in out REF_CALL_PACKET);
    procedure SEND (A: in REF_ANSWER PACKET);
    procedure RECEIVE (A: out REF_CALL_PACKET);
    procedure CLOSE;
  end CALLEE;

private
  type RETURN ADDRESS is new ADDRESS_STRUCT;
  type HEADER TYPE is new NATURAL;
end STANDARD_COMM INTERFACE;
```

Fig. 1: Standard Communication Interface

In doing this, it recognizes which entry calls are really remote after the VN-to-PN mapping, and transforms them according to the defined protocol.
The implementation of remote rendezvous is very similar to that of remote procedure calls [Nelson 81].
Each callee PN is provided with an entry port task, managing incoming calls for the interface tasks of the VNs allocated onto the PN. Such task routes the call (i.e. the operating system message) to the right addressee, by dynamically instantiating local agents performing the local rendezvous and send the parameters back to remote callers.
Each remote entry call at the caller side is transformed into a call to a procedure which prepares and sends the message to the remote callee PN and waits for the answer (out parameters, time-out, propagated exceptions).

All data types, procedures and tasks supporting the remote rendezvous protocol are introduced by the automatic transformation tool, and rely upon the SCI.

The tools have been implemented in Ada and can be ported with limited effort: there are minor well-identified portions which depend on the compiler or operating system features (Ada library organization, file system structure).

6. FMS: a case study

The described methodology and tools have been developed within the DIADEM MAP project, partially funded by the Commission of European Communities. Testbed applications were developed, aiming to evaluate the effectiveness of the approach for distributed systems.

More recently, the VN approach has been used for a real-world case study, concerning the development of a Flexible Manufacturing System (FMS) software control system.

An FMS requires the coordination of soft-real time activities performed in different cells and the implementation of optimization strategies at different levels. Different activities are grouped into:

- a planning level, responsible for production planning and scheduling;
- an operation management level, responsible for the coordination and monitoring of the whole plant in order to meet production requirements;
- a process control level, which is related to data collection and control of single production units (and where real-time constraints can be considerably harder).

The case study started from the assumption that at least the first two levels could meaningfully benefit by an Ada implementation, whereas the lowest level is left to typical special-purpose machines (Programmable Logic Controllers). Nevertheless, the use of different machines (e.g. single-board computers) could suggest the use of Ada for this level too.

The case study was chosen amongst on-going developments based on the so-called FMOS (Flexible Manufacturing Operating System) approach [Corti et al. 87], that models the operation management level as an operating system, considering machines as resources, the dispatcher as an OS scheduler, and the pieces to be worked as jobs, each requiring a certain amount of resources, described in its so-called "technological route".

Therefore the design of the FMS resulted into the main following VNs:

- strategic planning VN: it produces relatively unconstrained monthly production plans taking due dates and plant capacity as inputs;
- tactical planning VN: it produces daily plans starting from the monthly plan and validates it using feedbacks coming from the operation management level (unit status, fulfillment of daily plan);
- dispatching VN: it decides when and which pieces to introduce into the plant according to the daily plan;
- mission manager VN: it is responsible for tool replacement on machining centers;
- machine handler VN: a generic VN representing an abstraction of a machine (e.g. machining center, washing machine, tester, warehouse, tool room, automatically guided vehicles) and coordinating its activities according to predefined policies (it interacts with the special purpose process control subsystems);
- task pallet VNTs: it describes the technological route of a particular pallet (a pallet carries either pieces which have to be processed in the same way or tools to be delivered to a given machine);
- process monitor VN: it performs plant real-time monitoring;
- system monitor VN: it performs off-line statistics and reports;
- database manager VN: a database is needed to store all relevant information on the plant status;
- user interface VN: it manages all user interfaces (different user levels exist from the line director to operators, each having a different "view" of the controlled plant);

The hardware system is made of four identical computers, in two pairs for reliability purposes, one essentially devoted to planning activities and the other responsible for the operation management activities. A workstation or a personal computer can be used for graphic I/O.

The life-spans of all these VNs are quite different. User interface, machine handlers, database manager, process monitor and dispatcher are always running, whereas planning and system monitor run at specific times.

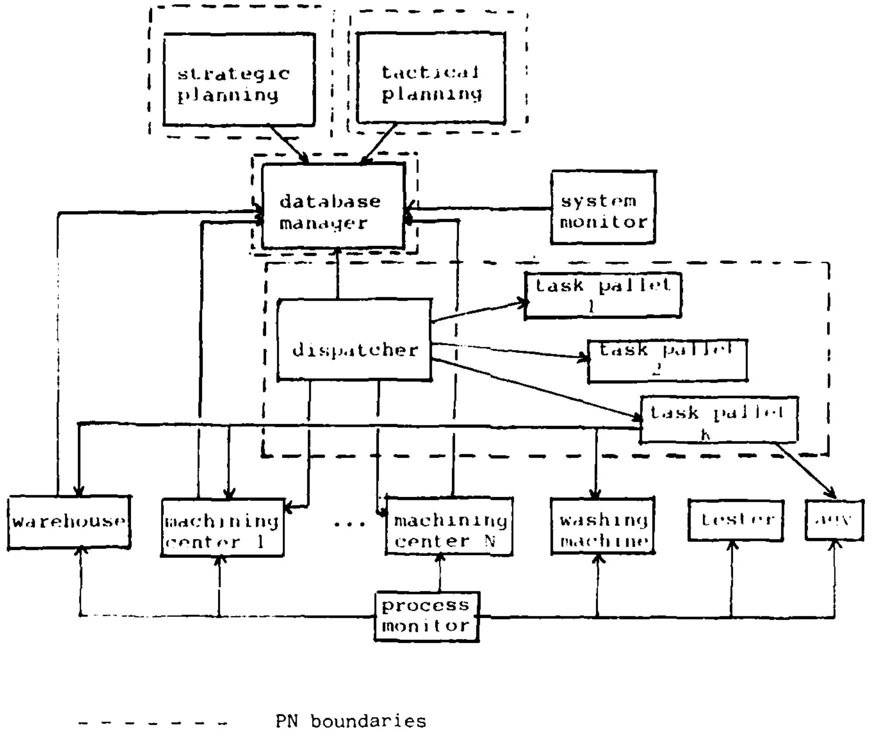


Fig. 2 Examples of VNs and PNs from the FMS case study

In the Ada VN solution, each machine handler VN contains an interface package providing the typical FMOS operations as entries:

- RESERVE: declares the need for the resource;
- LOAD: asks the machine to load the pallet carrying the piece;
- UNLOAD: asks the machine to unload the pallet;
- WORK: asks the machine to process the pallet;
- RELEASE: cancels a reservation.

The code of a task_pallet describes the technological route in terms of calls to the above operations (entries).

Different configurations are possible:

- The system is relatively static: all on-line VNs are grouped into a single PN, ensuring the efficiency of inter-VN communication (that would be performed as local rendezvous), while "batch" VNs are assigned to different PNs, in order to allow separate activation.
- Some of the on-line VNs could be changed during the system lifetime: they must be designed as VNTs and mapped onto separate PN to allow their *reconfiguration*.
- new or modified technological routes are likely in the future: each task_pallet VNT instance will correspond to a a different PN, with lots of remote communication.

Figure 2 shows some VNs and their interactions (tool management is not considered). The dotted lines identify some PNs of a possible configuration: all inter-VN communications represented by arrows which cross PN boundaries are processed by the transformer.

The Ada implementation is going to be completed in a few months, but some results can already be compared with the parallel industrial development (which adopts the multiprogramming approach with Pascal).

The monoprogramming approach has improved the development a lot, by providing much stronger interface checking, allowing a more natural modelling of the whole system, giving enough flexibility to delay many critical design decisions (like subsystem allocation). Moreover, although FMOS was already a reuse-oriented technology, the production of reusable code has been meaningfully improved.

It is worth noting that there are many similarities between FMS and C3I systems, even if sizes and application requirements differ. Both use host-based systems, present real-time features and perform many different activities, like user interface management, sensor management and processing, information collection and management.
Therefore, VN approach seems to be a good candidate for C3I developments.

7. Conclusions

Our assumption that a monoprogramminq approach is very effective one for host-based systems has been confirmed by the presented case study.

This methodology, that was originally developed for embedded distributed system programming, turns to be a potential key point for major improvements in future host-based developments.

Both method and tools need some refinements. First of all, the VN approach has an object-oriented flavour which should be exploited, investigating the possibility of an integration with already existing OOD methods. Furthermore, the approach proved to be too flat: it should be possible to define VN hierarchies in order to achieve a better structuring.
The tools are at a prototype stage, even if they have been partially engineered during the case study, and a need for stronger integration with other APSE tools (compiler, debugger, library manager) has been perceived by the programmers.

There is also a more substantial objection to the monoprogramming approach. Is it really cost-effective to constrain a huge software system to be developed as a single Ada program? Actually we feel that a break-even point exists over which the multiprogramming approach is mandatory. The convenience essentially depends on the nature of the system: huge, highly heterogeneous systems are likely to require multiprogramming, and perhaps different implementation languages for different subsystems, whereas more integrated systems can be more effectively managed in a monoprogramming style. Nevertheless future improvements and assessments of the VN approach, supporting multiple programming paradigms (including AI techniques), could move the break-even point ahead.

8. Acknowledgements

The DIADEM project provided the inspiration and the most material for this paper. I would like to thank our partners of GEC Software (UK) and Imperial College of Science and Technology (UK). I would also like to thank all FMOS designers.

REFERENCES

[Corti et al. 87]
P.L.Corti, F.Maderna, A.Mollo, M.Saccaggi, "How to make software reusable: the FMOS approach", Proc. Int. Symp. on Automotive technology and Automation, October 1987.

[Dapra et al. 84]
A.Dapra`, S.Gatti, S.Crespi-Reghizzi, A.Dapra`, F.Maderna, D.Belcredi, A.Natali, R.A.Stammers, M.D.Tedd, "Using Ada and APSE to Support Distributed Multimicroprocessor Targets, Ada Letters III(6), pp.57-65.

[Goldsack et al. 87]
S.J.Goldsack, C.Atkinson, A.Natali, A.Di Maio, F.Maderna, T.Moreton, "Ada for Distributed Systems - A Library of Virtual Nodes", Proc. Ada-Europe Conf., Stockholm, The Ada Companion Series, Cambridge University Press, 1987, pp. 253-265.

[Moreton et al. 87]
T.Moreton, J.C.D.Nissen. S.Desai, C.Atkinson, A.Di Maio, "Tools for the Building of Distributed Ada Programs", Proc. Ada-Europe Conf., Stockholm, The Ada Companion Series, Cambridge University Press, 1987, pp.266-278.

[Nelson 81]
B.J.Nelson, Remote Procedure Call, PhD Thesis, dept. of Computer Science, Carnagie-Mellon University, May 1981.

[Tedd et al. 84]
M.D.Tedd, S.Crespi-Reghizzi, A.Natali, Ada for Multi-microprocessors, The Ada Companion, Series, Cambridge University Press, 1984.

[WRTAI 87]
"Proc. of International Workshop on Real-Time Ada Issues", Ada Letters VII(6), Fall 1987.

CONCURRENT PROCESSING TECHNIQUES APPLIED TO SIMPLE LINE DRAWINGS

Reginald L. Walker

Hampton University
Hampton, Virginia 23668

ABSTRACT

Concurrent programming may change the underlying structure of many programming applications. The algorithms are being restructured to incorporate the new concurrent programming techniques. These new algorithms are being developed by examining the sequence of execution for a particular application and enhancing these execution sequences by applying concurrent programming techniques.

INTRODUCTION

The application area chosen for this research[1] is graphics processing; in particular, the generation of line drawings. This area consists of several dependent and independent tasks that are used to generate different graphical images, such as the algorithms which are used to generate xy-coordinates for line drawings, object transformations, computer simulations using line drawings, etc. Ada was chosen as a programming tool because the language contains language defined concurrent programming constructs. Ada and concurrent techniques are applied to some commercial and in-house application algorithms.

OBJECTIVE AND GENERAL IMPLEMENTATION

In this research, the main goal is to study the structure of the underlying routines of several graphics packages and develop concurrent algorithms for selected graphics applications. The initial implementation of each general algorithm incorporates a study of the subprograms that are necessary for the generation of graphical images. The process is iterative and consists of the implementation, testing, recording resulting execution times, and restructuring of the programming code. The process continues until optimal execution times are obtained for each general algorithm.

Several unique features of the Ada language were used in the generation of code for algorithms and device drivers. The "package" feature was used to create modular code and to minimize the amount of code requiring recompliation during the iterative search for an optimal sequential implementation. Run-time errors were minimized through the use of the range constraint feature provided by the language.

Another feature of the Ada language used in this study was the pragma. The INLINE pragma was used to create a sequential program with a limited number of external function and procedure calls. This pragma causes executable file sizes to increase because the actual code for each function and procedure call is inserted where the actual subroutine call is located. Since the programs are executed on a sequential machine and file sizes are not a constraint, this pragma proved to be very beneficial in the execution of the sequential implementations. Also, this pragma strengthens the use of modular structuring for the general algorithm and the device driver. The other construct provided by the Ada compiler is the OPTIMIZE pragma. This pragma instructs the compiler to optimize the executable code.

[1] This research is being funded under NASA Grant Number NAG-1-789, NASA Langley Research Center.

ENHANCEMENT OF DEVICE DRIVERS

Concurrent programming techniques are not limited to the concurrent implementation of the algorithms, but is also applied to the device drivers. Applying concurrency to the device drivers reduces the execution time required by the graphics terminal. For sequential implementations of graphic applications, the processor in the graphics terminal is not always adequate for the timely execution of programs. After the implementation of several sequential programs for a variety of graphics applications, the underlying routines for the device driver are studied to determine several concurrent implementations for each of the underlying routines. With the enhanced device driver, the generation of graphical images is faster.

APPLICATION ALGORITHMS

The Tiling-Explorer Algorithm used is an algorithm that generates a large variety of unpredictable patterns which appear symmetrical. Due to the slow rate of execution and image generation, this algorithm is a good candidate for this research. Execution of the implemented algorithm varies based on the values that are used for the paremeters RES, BETA1, BETA2, GAMMA, ALPHA, and MODF. In order to perform the initial studies using the algorithm, the paremeters are constants during the generation of the graphical images.

The following is the Ada implementation of the Tiling-Explorer Algorithm that generates the image file.

```
-- Needed "WITH" statements go here.
procedure EXPLORER is

FILE_NAME        : string (1..6);
COLOR, M         : integer;
CHAR             : character;
FILE_PAREMETERS: file_type:
BETA1: integer := -150;
BETA2: integer := -209;
GAMMA: integer := 877;
ALPHA: integer := 100;
MODF : integer := 519;

procedure SET_UP is
begin
   SELECT_CODE(0);
   DA_ENABLE(1);
   DA_LINES(32);
   DA_VISIBILITY(1);
end SET_UP;

procedure TILING_EXPLORER is
begin
   DA_VISIBILITY(0);
   CLEAR_DIALOG;
   DA_LINES(30);
   SELECT_CODE(2);
end RE_SET;

procedure TILING_EXPLORER IS
   C,R,Q: integer:
   X,Y  : integer;
   Z    : float;
begin
   SGOPEN(1);
   for Q in 0..4095 loop
      if Q mod 10 = 0 then
         SGUP;
      end if;
      for R in 0..3071 loop
         X:= BETA1 + GAMMA * Q;
         Y:= BETA2 + GAMMA * R;
         Z:= float(ALPHA) * sin(float(X))
            + sin(float(Y)));
         C:= FLOAT_TRUNCATION(Z);
         if C mod MODF = 0 then
            MARKER (Q,R);
         end if;
      end loop;
   end loop;
   SGCLOSE:
end TILING_EXPLORER;

procedure CHAR_CONVERT is
begin
   if M = 0 then
      CHAR:= '0';
   elsif M = 1 then
      CHAR:='1';
   elsif M = 2 then
      CHAR:= '2';
   elsif M = 3
      CHAR:= '3';
   elsif M = 4
      CHAR:= '4';
   elsif M = 5 then
      CHAR:= '5';
   elsif M = 6 then
      CHAR:= '6';
   elsif M = 7 then
      CHAR:= '7';
   elsif M = 8 then
      CHAR:= '8';
   elsif M = 9 then
      CHAR:= '9';
   end if;
end CHAR_CONVERT:

procedure CREATE_NAME is
   YYY,XXY: integer;
begin
   if M . 10 then
      CHAR_CONVERT;
      FILE_NAME(1):= CHAR;
      FILE_NAME(2):= 'a';
```

```
         else
            YYY:= M;
            XXY:= M mod 10;
            M:= (M - XXY)/10;
            CHAR_CONVERT;
            FILE_NAME(1):= CHAR:
         end if;
      end CREATE_NAME;

      procedure TILING is
      begin
         put(FILE_PARAMETERS,"  #  ");
         put(FILE_PARAMETERS,M);
         new_line(FILE_PARAMETERS);
         BETA1:= -150 + 2 * M;
         BETA2:= -209 + 3 * M;
         GAMMA:=  877 - 5 * M;
         ALPHA:=  100 - M;
         MODF :=  519 - 5 * M;
         put(FILE_PARAMETERS,BETA1);
         put(FILE_PARAMETERS,", ");
         put(FILE_PARAMETERS,BETA2);
         put(FILE_PARAMETERS,", ");
         put(FILE_PARAMETERS,GAMMA);
         put(FOLE PARATEMERS,", ");
         put(FILE PARAMETERS,ALPHA);
         put(FILE PARAMETERS,", ");
         put(FILE PARAMETERS,MODF);
         new_line(FILE_PARAMETERS,2);
      end TILING;
   begin
      open(FILE_PARAMETERS,
           in_file,
           "TILING_PARAMETERS.APP");
      reset(FILE_PARAMETERS);
      get(FILE_PARAMETERS,M);
      FILE_NAME(3..6):= ".APP";
      TILING;
      CREATE_NAME
      create(GENERATE_FILE);
      set_line_length(GENERATE_FILE,80);
      set_page_length(GENERATE_FILE,0);
      MARKER_TYPE(0);
      SET_UP:
      COLOR:=M mod 15;
      if COLOR = 0 then
         COLOR:= COLOR + 2;
      elsif COLOR = 1 then
         COLOR:= 7;
      end if;
      LINEINDEX(COLOR);
      TILING_EXPLORER;
      RE_SET:
      close(GENERATE_FILE);
      reset(FILE_PARAMETERS);
      put(FILE_PARAMETERS,0);
      close(FILE_PARAMETERS);
   end EXPLORER;
```

The following are execution times for the TILING_EXPLORER Algorithm that was studied.

```
Executable file     CPU time
case 1
TILING_EXPLORER    1:00:01.98

case 2
(w/INLINE pragma)

TILING_EXPLORER    0:53:14.81

case 3
(w/OPTIMIZE pragma)
TILING_EXPLORER    1:00:31.42

case 4
(w/ONLINE pragma and OPTIMIZE
 pragma)
TILING_EXPLORER    0:53:18.47

case 5
(compiled using
 ADA/OPTIMIZE=TIME)
TILING_EXPLORER    1:00:13.62

case 6
(w/LINE pragma and
 compiled using
 ADA/OPTIMIZE=TIME)
TILING_EXPLORER    0:53L18.89

case 7
(The FOR LOOP constraints were switch.
 The outer loop performed 3072
 inerations and the inner loop
 performed 4096 interations.)
TILING_EXPLORER    0:53:56.38
```

At this point in the research, the file sizes did provided an added constraint. If the file sizes were a problem, the optimal execution time of 53 minutes and 14.81 seconds could not have been achieved. It was also apparent for this particular graphics package, that the OPTIMIZE pragma produced added overhead in all of the implementations using this pragma. In this graphics package, there was a total of 12,682,498 function or procedure calls(This number includes only the user defined subroutine calls). In images that are more dense plots, the number of subroutine calls increases.

IMPLEMENTATION ENVIRONMENT

The implementation environment includes Tektronic Graphics Terminals 4107 and 4109, and the host computer is the VAX 11/780. The Ada compiler used in this study is Digital's VAX/Ada.

FUTURE IMPLEMENTATIONS

The future implementation of the graphics packages will be hosted by a VAX 8300 series computer using the concurrent implementations of the optimal sequential executions as benchmark. Future implementations may also be implemented on different computer hosts and using different programming languages.

SUMMARY

General algorithms of the type that have been implemented in this research provide a good introduction to the study of the impact of concurrent programming on the performance of graphics packages. A further study of this algorithm and similar algorithms will lead to the implementation of concurrent programs for these simple graphics applications.

REFERENCES

1. 4106/4107/4109/CX Computer Display Terminals, TEK Programmers Reference Manual, Tektronix, Inc., Beaverton, Oregon, 1984.

2. Developing Ada Programs on VAX/VMS, Digital Equipment Corp., Maynard, Mass. 1985.

3. Pickover, Clifford A., "Blooming Integers", Computer Graphics World, March 1987.

4. VAX Ada Language Reference Manual, Digital Equipment Corp., Maynard, Mass., 1985.

5. VAX Ada Programmer's Run-Time Reference Manual, Digital Equipment Corp., Maynard, Mass., 1985.

Reginald L. Walker received the B.S. Degree in Mathematics from Morris Brown College, Atlanta, Georgia, in 1981. He also holds the M.S. degree in Mathematics from Atlanta University, Atlanta, Georgia, received in 1986. He is presently an Instructor of Computer Science at Hampton University, Hampton, Virginia.

SATISFYING EMERGENCY COMMUNICATION REQUIREMENTS WITH DYNAMIC PREFERENCE CONTROL

Tzilla Elrad
Illinois Institute of Technology
Department of Computer Science
Chicago, IL 60616
(312) 567-5142

Fred Maymir-Ducharme
AT&T Bell Laboratories
IH 6U-204
Naperville-Wheaton Rd.
Naperville, IL 60566
(312) 979-2290

ABSTRACT:

Emergencies and crises must be dealt with immediately and gracefully by communications software. Concurrent, distributed environments must rely on message passing for communication and synchronization between processes and sometimes between processors. Recent concurrent programming languages exploit the processing power of parallel systems by the use of nondeterministic constructs. But military communications software requires strict control of nondeterminism in order to efficiently and immediately handle emergencies and crises that must be anticipated and correctly dealt with. Although many programming languages contain different controls on nondeterminism, none contain a construct equal to dynamic preference control [5] [7]. Dynamic preference control has the flexibility and expressive power needed by military communications to best handle emergencies and crises at the programming language level. Preference control allows the programmer to explicitly give preference to the detection of emergency situations and thereby responding to the crises immediately.

This paper discusses the added explicit expressive power of using dynamic preference control and its many applications to military communications software. One possible example may be a weapon with several defensive and offensive operations. During normal operation, the offensive operations may be given preference. But in case of an attack, the defensive operations must be given preference to help prevent the destruction of the weapon. These situations will change dynamically and mandate that the software also change its preferences dynamically.

KEY WORDS AND PHRASES:

Emergencies and crises requirements, military communications software, Ada, controlling nondeterminism, parallel processing, dynamic preference control.

1. INTRODUCTION

Emergencies and crises must be dealt with immediately and gracefully by communications software. Concurrent, distributed environments must rely on message passing for communication and synchronization between processes and sometimes between processors. Recent concurrent programming languages exploit the processing power of parallel systems by the use of nondeterministic constructs. But military communications software requires strict control of nondeterminism in order to efficiently and immediately handle emergencies and crises that must be anticipated and correctly dealt with. Although many programming languages contain different controls on nondeterminism, none contain a construct equal to dynamic preference control [5] [7]. Dynamic preference control has the flexibility and expressive power needed by military communications to best handle emergencies and crises at the programming language level. Preference control allows the programmer to explicitly give preference to the detection of emergency situations and thereby responding to the crises immediately.

Dynamic preference control allows the programmer to assign expressions as values to preferences instead of assigning them constants; these preferences can dynamically change as conditions change (e.g. emergencies or different modes of operations). Preference control allows the programmer to explicitly give some alternatives within the nondeterministic construct (e.g. the *select statement* in ADA) the chance to be considered before others with a lower preference. Real-time military communication software requires the dynamic preference construct to better control nondeterminism during emergency and crisis situations. This construct adheres to software engineering principles and is very well suited for military communications.

Our paper contains some examples of communication software requiring dynamic preference control to efficiently handle emergencies in the proper timely fashion. First, in section two, we briefly cover the classification of different controls on nondeterminism in current concurrent programming languages. The select statement is the nondeterministic construct in Ada. Examples of these controls in Ada are illustrated in [5], [6] and [7]. Section three then discusses the need for preference control during parallel processing. This is followed by a description of the syntax and semantics of the dynamic preference control construct **pref**. Finally, in section four we describe a few applications of dynamic preference control to hypothetical military software to meet emergency and crisis requirements. An example is then illustrated and analyzed using software engineering principles.

2 CONTROL CLASSIFICATIONS

2.1 PRIVATE CONTROL

Private control: nondeterminism restricted by variables local to the task. Private control is considered open if the boolean expression is true; it is closed otherwise. Only alternatives with open private control are considered for selection by the nondeterministic construct.

2.2 CONSENSUS CONTROL

Consensus control: nondeterminism restricted by environmental/communication constraints, which we classify as consensus. The choices are restricted to only those for which the communication request for the entry has been queued. Consensus control is considered established if the rendezveous can be established.

2.3 HYBRID CONTROL

Hybrid control: nondeterminism restricted by both local boolean and environmental constraints (private control and consensus control), which we classify as hybrid control. Hybrid control is available if private control is open and consensus control is established.

An alternative is ready if all of the controls have been satisfied; that is: if the private control is open and/or if the consensus control is established or if the hybrid control is available. The nondeterministic construct will check all of the controls in each alternative and then choose one of the ready alternatives. In our experience, the programmer requires a combination of these three controls when using the nondeterministic construct.

2.4 OTHER RELATED CONSTRUCTS

Many concurrent programming languages contain all or some of these controls, but have never classified them. CSP [13] [14] contains private, consensus and hybrid control. Ada [16] only allows consensus and hybrid control; although private control can be implemented combining the **when** construct with the **delay** alternative. And Concurrent C, which is based on Ada, allows private, consensus and hybrid control. The **by** construct was introduced by Andrews [2] and Concurrent C [11] later implemented the **by** construct and the **such that** construct. These constructs allow the entry to selectively choose an entry call from the entry queue instead of receiving the entry calls from the queue in FIFO order, which is the default. Concurrent C and Ada have implicitly defined preferences built into the language; in Concurrent C, an available accept alternative is chosen before an open immediate alternative. None of these languages contain a primitive to control preferences explicitly; this led us to the implementation of the "pref" primitive, which is necessary for preference control. The **by** and the **such that** constructs differ from preference control in that they control choices from with an entry's queue; "pref" controls the preferences of entries and other alternatives within the *select statement*. The "cell" concept [17] allows explicit preference control amongst the entries within the *select statement* by attaching labels to entries and ordering the labels statically.

3 THE PREFERENCE CONTROL CONSTRUCT

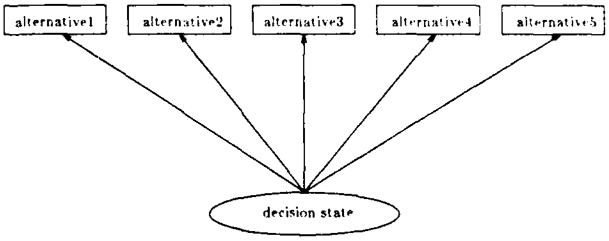
3.1 THE NEED FOR PREFERENCE CONTROL

The extension of Ada to include static or dynamic preference control in the nondeterministic construct, the *select statement*, would greatly increase the programmers' explicit expressive power. Preference control is an implementation of the Branching Wide Horizons Principle, which is our extension of the known Wide Horizon Principle by N. Francez and S. Yemini [9]. The Wide Horizon Principle describes the need for a nondeterministic construct in concurrent programming languages; whereas the Branching Wide Horizons Principle extends this need to include preference control.

3.1.1 THE WIDE HORIZON PRINCIPLE The Wide Horizon Principle [9] : "Whenever the semantics of a construct C in a language for concurrent programming implies the delay of a process (task) executing C, C should be able to have other alternatives, and all such constructs should be able to serve as alternatives to each other. "

Example 1

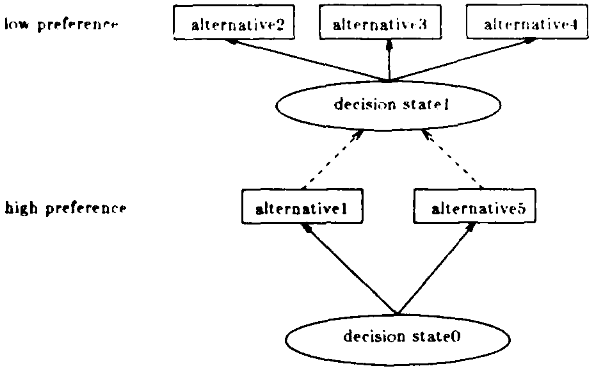
A diagram of the wide horizon principle



3.1.2 THE BRANCHING WIDE HORIZONS PRINCIPLE The "Wide Horizon Principle" supplies many alternatives to choose from nondeterministically, but it also allows the choosing of a less urgent alternative. We need to further control this choice by specifying the relative urgency of each alternative; this is accomplished via preference control. The Branching Wide Horizons Principle describes the need for the preference control construct in concurrent programming languages.

The Branching Wide Horizons Principle Whenever the semantics of a construct S, which implements the wide horizon principle, implies the delay of S (e.g. the delay of all of the alternative constructs C in S), S should be able to have a finite set of less preferable horizons.

Example 2
A diagram of the branching wide horizons principle



3.2 STATIC PREFERENCE CONTROL

Before proceeding, the difference between priority and preference control must be distinguished. The term priority is used to denote a race amongst different tasks. We need to control the race amongst the different alternatives within a select statement in a task. The term preference denotes the race amongst alternatives and differentiates this race from task priorities.

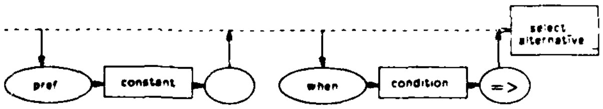
We have classified preference control as nondeterminism restricted by preferences: preference control gives the programmer the power to assign preferences to the choices within the nondeterministic construct. Each entry inside a nondeterministic construct may (but need not) have a preference, which is a constant of the predefined subtype **pref**. A lower value indicates a lower degree of urgency: the range of preferences is implementation defined. (e.g. a ready entry with a preference = 2 is chosen before a ready entry of preference = 1).

Extended Syntax for the Ada select alternative with static **pref**:

Example 3

pref <constant>: **when** <condition> => **accept** <entry>

SELECT ALTERNATIVE:



The preference control construct supports the Branching Wide Horizons Principle. Static preference control entails assigning a constant value to entries statically. These values can be stored at compile time and cannot change during run time. (See example 4)

Preference control should appear as part of the controls on nondeterminism within the *select statement* (e.g., as part of the entry guard); therefore it must be included as part of the task body. A complete guard should include preference control, private control and consensus control. We also chose to explicitly state the preference within the task specification since it may be part of the requirements, and not just part of the implementation.

Example 4

EXAMPLE IN ADA WITH THE STATIC PREFERENCE CONTROL PRIMITIVE

```
type urgencies is ( low, medium, critical )
-- these enumerated constants are statically
-- assigned to each entry.

task CONTROLLER is
 pref critical   : entry ENTRY_A(..);
 pref medium  : entry ENTRY_B(..);
 pref low        : entry ENTRY_C(..);
end CONTROLLER;

task body CONTROLLER is
begin
  loop
   select
     pref urgent: accept ENTRY_A(..) do
         ACTION_A(..); end ENTRY_A;
   or
     pref medium: accept ENTRY_B(..) do
         ACTION_B(..); end ENTRY_B;
   or
     pref low: accept ENTRY_C(..) do
         ACTION_C(..); end ENTRY_C;
    end select;
   end loop;
end CONTROLLER;
```

Semantics of preference control construct Select one of the available alternatives with the highest preference value. More than one entry can have the same pref value; this will increase the nondeterminism of the choice.

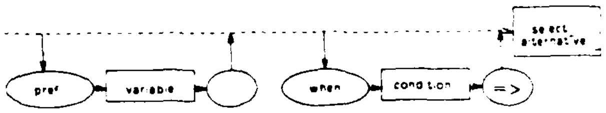
3.3 DYNAMIC PREFERENCE CONTROL

Extended Syntax for the Ada select alternative with dynamic **pref**:

pref <variable>: **when** <condition> => **accept** <entry>

Example 5

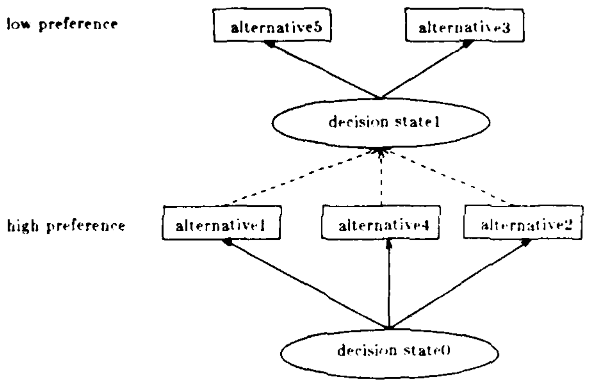
SELECT ALTERNATIVE:



Dynamic preference control allows the assignment of values to the variables associated with each entry, and allows them to change dynamically. Dynamic preference control is much more powerful than static preference control.

This diagram illustrates the power of dynamically changing the preferences from those illustrated in example 2

Example 6



Example 7

EXAMPLE IN ADA WITH THE DYNAMIC PREFERENCE CONTROL PRIMITIVE

```
type urgencies is ( low, medium, critical )
pref_A, pref_B, pref_C :urgencies;
-- In this case, the "pref_ " variables can dynamically
-- be changed by being assigned new values; thereby
-- changing preferences.

task CONTROLLER is
 pref pref_A : entry ENTRY_A(..);
 pref pref_B : entry ENTRY_B(..);
 pref pref_C : entry ENTRY_C(..);
end CONTROLLER;

task body CONTROLLER is
begin
(pref_A,pref_B,pref_C) :=
    ( critical,medium,low )
  --INITIALIZE PREFERENCES
  loop
   select
    pref pref_A: accept ENTRY_A(..) do
        ACTION_A(..); end ENTRY_A;
        pref_mod(pref_A,pref_B,pref_C) := (...)
    or
    pref pref_B: accept ENTRY_B(..) do
        ACTION_B(..); end ENTRY_B;
        pref_mod(pref_A,pref_B,pref_C) := (...)
    or
    pref pref_C: accept ENTRY_C(..) do
        ACTION_C(..); end ENTRY_C;
        pref_mod(pref_A,pref_B,pref_C) := (...)
   end select;
  end loop;
end CONTROLLER;
```

NOTE: "(...)" represents a vector of values assigned to the vector of preferences. This can be represented by an array. See example 8.

To give a clearer illustration of the preference control construct, examples four and seven did not include private control, which is implemented using the **when** construct. You'll notice that preference control appears in both the task specification and in the task body. Preference control may sometimes be part of the requirements, not only part of the solution; therefore it belongs in the task specification. Preference control is also one of the controls on nondeterminism and belongs in the task body, alongside the other controls. Hence, our syntax allows this necessary duplication.

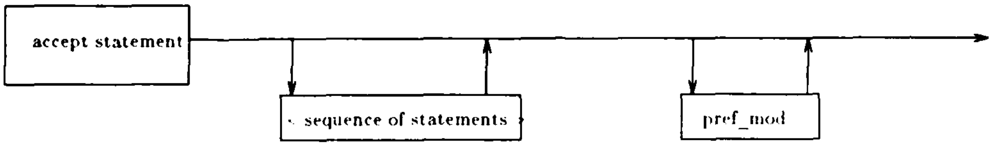
The Use of Zero Preference:

For compiler optimization, dynamic preference control also allows the assignment of "0" to an entry's preference variable. **"pref 0:"** indicates to the compiler that the associated entry should not be considered for selection until the preference is changed. As in the example of the "Readers Writers Problem", in which all writers must be pre-empted until the readers are through reading, the **accept writer** entries can be assigned **pref 0:** to discourage the compiler from needlessly evaluating these entries' guards until it becomes feasible.

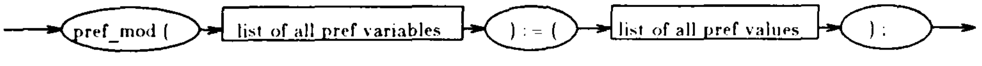
Where, When and How to Update Dynamic Preference Control Variables

The initialization of the preference control variables *must be done within the task in which the select statement* resides. Afterward, assignments to the **pref** variables must be done through the preference modification structure - **pref_mod**. The use of the **pref_mod** structure may only appear after the body on an accept alternative. The extended syntax for the *accept alternative is as follows:*

Example 8



pref_mod :



pref_mod (<list of all pref variables>) := (<list of all pref values>)

(See examples 7 and 10)

The preference modification structure simultaneously changes all of the preference variables to ensure the proper preferential relationship between the alternatives. Modifying only one preference variable may not ensure the intended relation of preferences; this is because the value of another preference variable may have changed and offset the previous intended preferential relation of alternatives.

4. USING DYNAMIC PREFERENCE CONTROL FOR MILITARY SOFTWARE

Communication is one of the most critical and expensive aspects of distributed software; therefore, efficiently controlling intertask communication is of outmost importance. There are many possible applications of preference control to military communication software. This paper focuses on the use of dynamic preference control to satisfy emergency communication requirements. Crises and emergencies require immediate attention and reaction by the software. A task that is continuously supporting some military operations must be able to stop normal operation and deal with any emergency situation in a real-time response time frame. An embedded system may contain a task whose responsibilities include receiving coordinates from several radars and plotting the results on a monitor for some other defensive or offensive task. This task may be required to give preference to accepting coordinates from the closest radars first, since they may pose the greatest danger. In the case of an emergency, in which one of the radars senses an attack, the information sent by that radar must be given preference and immediately processed to deal with the crisis.

Another example may be a telecommunications software system used by a military base. We will assume that this system is shared by all personnel of different ranks. Under normal conditions, all of the users should be given equal preference by the system when attempting to make phone calls. But an emergency situation may require that the system give a higher preference to serving the officials with the highest rank before attempting to serve other users in order to allow the highest level commands to be telephoned first. In this case, dynamic preference control would be necessary to control the otherwise nondeterministic choice.

To better illustrate the expressive power of the dynamic preference control construct, lets consider a military weapon with several defensive and offensive operations. We will assume that the software requirements specified that all operations by this weapon be centralized in one task. During normal operation, the offensive operations may be given preference. But in case of an attack, the defensive operations must be given preference to help prevent the destruction of the weapon. These situations will change dynamically and mandate that the software also change its preferences dynamically.

In the following example, DEFENSE_A, DEFENSE_B and DEFENSE_C represent different defensive operations which protect the weapon by managing and calling different types of shields, each of which may cover separate sections of the SECRET_WEAPON.

OFFENSE_A, OFFENSE_B and OFFENSE_C represent different offensive operations, which make calls to weapons that use different types of secret ammunitions. The task, SECRET_WEAPON, makes calls external to itself to increase concurrency and allow a faster response to other calls.

Example 9
PART OF THE SECRET WEAPON SOFTWARE WITHOUT DYNAMIC PREFERENCE CONTROL

```
task SECRET_WEAPON is
 entry DEFENSE_A(..);
 entry DEFENSE_B(..);
 entry DEFENSE_C(..);
 entry OFFENSE_A(..);
 entry OFFENSE_B(..);
 entry OFFENSE_C(..);
end SECRET_WEAPON;

task body SECRET_WEAPON is
loop
 select
   when SHIELD_INTACT[A] =>
        accept DEFENSE_A(..) do
        -- call or activate shield  A
        end DEFENSE_A :
  or
   when SHIELD_INTACT[B] =>
        accept DEFENSE_B(..) do
        -- call or activate shield  B
        end DEFENSE_B :
  or
   when SHIELD_INTACT[C] =>
        accept DEFENSE_C(..) do
        -- call or activate shield  C
        end DEFENSE_C :
  or
   when  AMMUNITION_EXISTS[A] =>
        accept OFFENSE_A(..) do
        -- call or activate Weapon  A
        end OFFENSE_A :
  or

   when  AMMUNITION_EXISTS[B] =>
        accept OFFENSE_B(..) do
        -- call or activate Weapon  B
        end OFFENSE_B :
  or
   when  AMMUNITION_EXISTS[C] =>
        accept OFFENSE_C(..) do
        -- call or activate Weapon  C
        end OFFENSE_C :
  else NULL;
-- this avoids raising an exception if
-- no alternative is ready

  end select;
 end loop;
end SECRET_WEAPON;
```

In example nine, SECRET_WEAPON simply loops forever, accepting any ready alternative nondeterministically. In the case of an emergency, which may require a specific defense and/or offense, this example cannot dynamically control the nondeterministic choice and specify the required alternatives for the specific crisis. Now consider the same example with dynamic preference control.

Example 10
PART OF THE SECRET WEAPON SOFTWARE WITH DYNAMIC PREFERENCE CONTROL

```
pref_vars is ( 4, 3, 2, 1, 0 );
EMERGENCY,D_A,D_B,D_C,O_A,O_B,O_C:pref_vars;

task SECRET_WEAPON is
 pref EMERGENCY : entry DYNAMIC_STATUS
         (A,B,C,D,E,F : in pref_vars);
 pref D_A : entry DEFENSE_A(..);
 pref D_B : entry DEFENSE_B(..);
 pref D_C : entry DEFENSE_C(..);
 pref O_A : entry OFFENSE_A(..);
 pref O_B : entry OFFENSE_B(..);
 pref O_C : entry OFFENSE_C(..);
end SECRET_WEAPON;

task body SECRET_WEAPON is
-- initialize preferences before entering loop
loop
 select
   pref EMERGENCY : accept DYNAMIC_STATUS
        (A,B,C,D,E,F : in pref_vars) do
          pref_mod (EMERGENCY,D_A,D_B,D_C,
          O_A,O_B,O_C) := (4,A,B,C,D,E,F)
  -- This entry maintains sole ownership
  -- for the preference on EMERGENCY
        end DYNAMIC_STATUS;
  or
   pref D_A : when SHIELD_INTACT[A] =>
        accept DEFENSE_A(..) do
        -- call or activate shield  A
        end DEFENSE_A :
  or
   pref D_B : when SHIELD_INTACT[B] =>
        accept DEFENSE_B(..) do
        -- call or activate shield  B
        end DEFENSE_B :
  or
   pref D_C : when SHIELD_INTACT[C] =>
        accept DEFENSE_C(..) do
        -- call or activate shield  C
        end DEFENSE_C :
  or
   pref O_A : when  AMMUNITION_EXISTS[A] =>
```

```
        accept OFFENSE_A(..) do
        -- call or activate Weapon  A
        end OFFENSE_A ;
 or
  pref O_B : when  AMMUNITION_EXISTS[B] =>
        accept OFFENSE_B(..) do
        -- call or activate Weapon  B
        end OFFENSE_B ;
 or
  pref O_C : when  AMMUNITION_EXISTS[C] =>
        accept OFFENSE_C(..) do
        -- call or activate Weapon  C
        end OFFENSE_C ;
 else NULL;
-- this avoids raising an exception
-- if no alternative is ready

  end select;
end loop;
```

A more intelligent and resilient program may check pertinent information after each operation and wish to change the dynamic preference control based on that evaluation. For example, the OFFENSE entries could have a check on the amount of ammunition remaining after each firing. If the ammunition is getting low for a specific entry, a higher preference could be given to the other OFFENSE entries in order to save enough ammunition to handle an emergency.

A comparison of examples nine and ten illustrates the advantages and added programming power of the dynamic preference control. The programmer can explicitly express a preferential order to the alternatives within the *select statement* and change the order dynamically as situations change. Under peacefull circumstances, one may wish to be fair and give all the alternatives the same preference to allow maintenance or testing. But in case of an attack, a higher preference could first be given to the defensive solutions until the opportunity arose to take the offensive and then give the offensive alternatives higher preference. Certain situations could arise requiring that some offensive alternatives be given higher preference than others, this is also possible with dynamic preference control.

Example 9, without preference control, cannot give different preferences to different alternatives, as would be required in case of emergencies or crises. Example 10 gives the programmer the explicit, expressive power to assign preferences and therefore meet emergency and crisis requirements.

5. *CONCLUSION*

Dynamic preference control can be used to meet military emergency and crisis requirements. The added explicit expressive power of dynamic **pref** allows the programmer to better control nondeterminism in a distributed environment in order to immediately and efficiently handle emergencies and crises. The power of parallel processing can be exploited by the use of nondeterministic constructs like the *select statement* in Ada. But this added power must be domesticated by understanding the available controls on nondeterminism and extending the language to allow dynamic preference control.

Static preference control can be implemented using existing language constructs [4] [5] [7] instead of the static preference control construct. But these implementations are very complicated, ambiguous and do not adhere to software engineering principles. Dynamic preference control, on the other hand, cannot be implemented using existing language constructs and without considerable overhead, which is unacceptable in real-time applications. The *select statement* must be extended to allow the **pref** primitive. We are currently extending a version of the Concurrent C compiler to allow dynamic preference control. (Concurrent C is very similar to Ada).

REFERENCES

1. G.R.Andrews, F.B.Schneider, "Concepts and Notations for Concurrent Programming" 1983 ACM 0010-4892/83/0300-0003.

2. G.R.Andrews, "Synchronizing Resources," ACM Trans. Programming Languages Syst. Vol. 3. Oct. 1981.

3. G.Booch, "Software Engineering with Ada" The Benjamin/ Cummings Publishing Co.,Inc 1983.

4. A. Burns, "Using Large Families for Handling Priority Requests," Ada LETTERS January, February 1987 Vol. VII, No. 1.

5. T. Elrad, F. Maymir-Ducharme, "Distributed Language Design: Constructs for Controlling Preferences" Proceedings of the 1986 International Conference on Parallel Processing in St. Charles, Illinois, August 19 - 22, 1986.

6. T. Elrad, F. Maymir-Ducharme, "Efficiently Controlling Communication in Ada Using Preference Control", Proceedings of the IEEE 1986 Military Communications Conference in Monterey, California, October 5 - 9, 1986.

7. T. Elrad, F. Maymir-Ducharme "Introducing the Preference Control Primitive: Experience with Controlling Nondeterminism in Ada", Proceedings of the 1986 Washington Ada Symposium in Laurel, Maryland, March 24 - 26, 1986.

8. T. Elrad, F. Maymir-Ducharme "Preference Control: A Language Feature for AIDA Applications," Proceedings of the 1987 Third Annual Conference on Artificial Intelligence & Ada, George Mason University, VA, October 14 - 15, 1987.

9. N. Francez, S. Yemini, "Symmetric Intertask Communication," ACM Transactions, Vol. 7 No. 4, 1985.

10. N. Gehani, "Ada: Concurrent Programming", Prentice Hall, 1984.

11. N. Gehani, W. Roome "Concurrent C*" AT&T Bell Laboratories, Murray Hill, New Jersey 07974, 1985.

12. M. Hennessy and G. D. Plotkin, "Full Abstraction for a Simple Parallel Programming Language" Proceedings 8th MFCS (1979), Lecture Notes in Computer Science 74, J. Becvar, Ed., Springer Verlag, 1979.

13. C.A.R. Hoare, "Communicating Sequential Processes" CACM 21,8, August 1978.

14. C.A.R. Hoare, "Communicating Sequential Processes," Prentice Hall International, 1985.

15. E. Horowitz, "Fundamentals of Programming Languages," Computer Science Press, 1984.

16. J.D. Ichbiah, et al, Reference Manual for the Ada Programming Language January 1983.

17. A. Silberschatz, "Cell: A Distributed Computing Modularization Concept", IEEE Trans. Softw. Eng., Vol. SE-10, No. 2, March 1984.

Fred A. Maymir-Ducharme received the B.S. degree in Computer Science from the University of Southern California in 1984, and the Ph.D. in Computer Science from the Illinois Institute of Technology in 1987, specializing in concurrent programming languages.

He is currently working on research and development at A T & T Bell Laboratories, in Naperville, Illinois. He is presently teaching a graduate class, "Software Engineering with Ada," at IIT. His areas of interest include multi-computers, parallel programming and distributed processing.

Tzilla Elrad, professor of Computer Science at the Illinois Institute of Technology, received the B.S. degree in Computer Science from the Hebrew University, the M.S. from Syracuse University in New York, and the Ph.D. in Computer Science from the Technion in 1982.

Tzilla's interests are in concurrent programmming language design, concurrent programming applications, testing and formal verification.

Why Strong Typing was added to DOD-STD-1838, The Common APSE Interface Set

Robert G. Munck

The MITRE Corporation, Bedford, MA. 01730
Munck@MITRE-Bedford.ARPA

1 Abstract

Most current Programmer Support Environments (PSEs) do not assign *types* to the pieces of data that they store or do so very weakly, for example through naming conventions. An effort is currently underway to upgrade DOD-STD-1838 (the CAIS) by adding strong typing to its data management system. "Strong typing" in this situation is a somewhat different concept from that of programming languages, mostly because the typed objects *persist* between executions of the system. It is therefore necessary to support *evolution* of the type definitions where a given tool may have to work correctly on old and new instances of a changed type.

This paper discusses the reasons strong typing is being added; these include the need for integrity of the data base of large programming projects, the desire to minimize human error and its effects, and the need of the DOD to move *all* of the information of large programming projects, including programs, test plans and data, documentation, the interrelations among the pieces of data, and the tools needed to manipulate the data, from the development contractor to the maintenance organization or another developer.

The paper also gives a brief description of how typing will be implemented in the revision, including an overview of the way type definitions are an explicit part of the resulting data structure.

2 Background

Early in the development of the Ada(R) language, it was recognized that a computer-based programming support environment was a practical necessity for Ada programmers. It has subsequently become clear that many of the promised advantages of Ada, including reusability of code and ease of maintenance, would require a level of commonalty in the *programming environment* similar to that of the language. The basic reason for this is that the product of an Ada programming project is not a single executable-form module as was often the case with earlier projects. Inter-related sets of source files, design documents, test plans, and many other kinds of information must also be present as part of the final product. Moreover, the *tools* that the original programmers use to store and manipulate these files and relationships among files must also be available to their successors who wish to reuse or maintain the product.

The concept of an Ada Programming Support Environment, or APSE, was well developed in the "Stoneman" document [STONEMAN80]. It specifies an architecture containing an identifiable interface between code having local host or operating system dependencies and code that would be portable from one host to another (see Figure 1). The code implementing this interface is called the Kernel APSE or KAPSE. The following text from the **CAIS Reader's Guide** [CAIS87] describes the DoD effort to develop a standard for that interface:

> When DoD started procuring tools for the Ada program, it did not restrict itself to procuring individual tools. Rather, the DoD embarked upon the procurement of APSEs. Two procurements were started: one by the Army, called the Ada Language System (ALS), and the other by the Air Force, called the Ada Integrated Environment (AIE). Unfortunately, the interfaces provided (by) the KAPSE ... were different in these two APSEs. Because of divergent approaches at the KAPSE interface level by the ALS and AIE contractors, a team was formed ... to define more specific KAPSE interface requirements. This team is the KAPSE Interface Team (KIT) and is chaired by (Patricia Oberndorf of the) Naval Ocean Systems Center (NOSC), a Navy laboratory. Added to the KIT was the KAPSE Interface Team from Industry and Academia (KITIA). The KIT/KITIA (produced) DOD-STD-1838, the Military Standard Common Ada Programming Support Environment (APSE) Interface Set (CAIS) [CAIS86].

The KIT/KITIA also developed a requirements document called the RAC (for Requirements and Criteria) [RAC86] that is in some ways a successor to STONEMAN. Unlike STONEMAN, it concentrated on the APSE interface seen by the KAPSE and discussed the needs of a DoD standard for this interface. It called loudly and clearly for *strong typing* of the data stored in an APSE. A Rationale document for the RAC [RAC87] was written to capture the discussions that led to it and flesh out its somewhat "specification-like" style. Parts of this paper are taken from that Rationale.

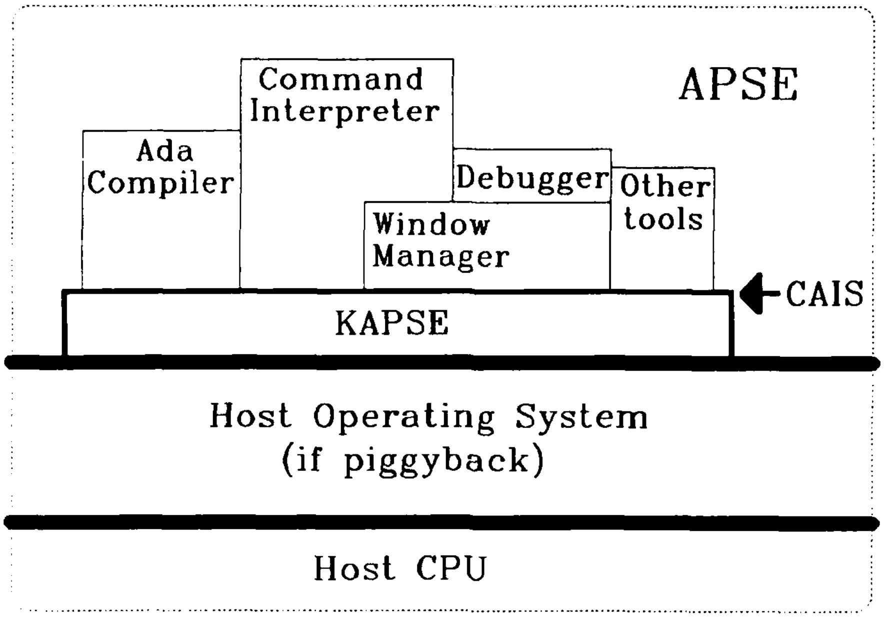


Figure 1 – System Architecture

A contractor was competitively chosen to revise the CAIS as indicated by the RAC, experience with the original CAIS and other APSEs, and advances in the state of the art since the work was begun. That revision, sometimes called CAISa, is about to enter the formal process needed to update *DoD-STD-1838* to DoD-STD-1838a; if it survives the review procedure, it should become the official standard sometime in early 1989.

3 Why Strong Typing?

The RAC section on data management begins with a small set of terse, highly-abstract requirements that set the stage for more specific discussions. This was done because the RAC Working Group (RACWG) of the KIT/KITIA wanted to state requirements independent of any particular implementation, but discovered that such statements tended to carry very different meanings for people from different specialty areas and backgrounds.

3.1 Abstract Data Management Requirements

The following are the "terse" data management requirements. As with the RAC Rationale, requirements from the RAC are printed in *italics* and followed by discussion. The reader's attention is drawn to the fact that most of the terse requirements are about typing in one way or another.

a. *There must be a means for retaining data*

b. *There must be a way of retaining relationships among and properties of data.*

c. *There must be a way of operating upon data, deleting data, and creating new data.*

Data might include the text of a piece of program source, test data, documentation, or a schedule. Data might also be the date a piece of test was created, the name of the author, information as to the piece of data from which some object code was compiled, or information as to which pieces of data contain other pieces of data.

These requirements are met in both CAIS and CAISa by an Entity - Relationship - Attribute (ERA) model in which entities and relationships are the nodes and edges of a general directed graph. ("Node" and "entity" are used interchangeably.)

Nodes are representations of real-world or conceptual entities such as files, directories, users, processes, and devices; *relationships* are associations from one node to another (uni-directional) or between two nodes (bi-directional); and *attributes* are named elementary values associated with nodes or relationships. See Figure 2. Attributes may be single values, arrays, or composites of various primitive types.

A *relation* is a set of relationships of the same type originating at one node -- a *one-to-many* association. Each relationship in a relation has its own set of attributes; there are no attributes for the relation as a whole. Certain of these attributes, called *keys*, can be used to select individual relationships from a relation. For example, the equivalent of a UNIX directory would be a node with a CHILD relation of relationships that have a key string attribute NAME. See Figure 3.

This is obviously an extremely flexible data structure; determining how to store the many kinds of data associated with a programming project will be a demanding and complex task. The major focus of the RAC and of the CAISa design is to add strong typing to the model.

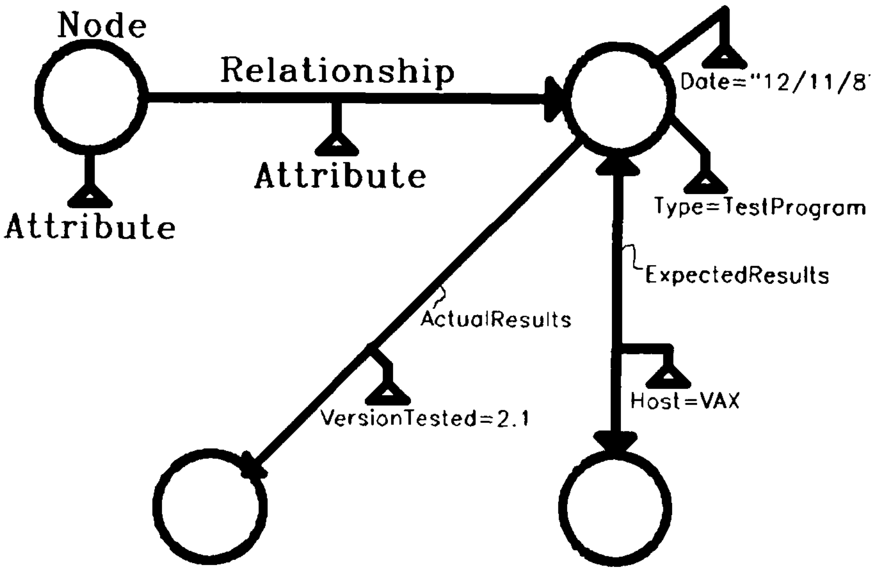


Figure 2 – Nodes, Attributes, Relationships

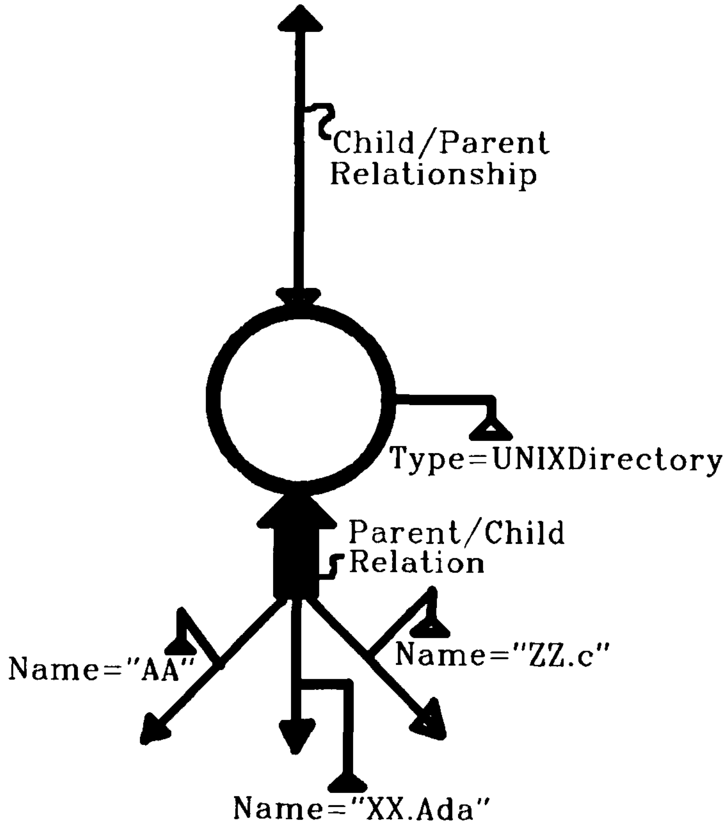


Figure 3 – Model of a UNIX Directory

d. There shall be a means for defining certain operations and conditions as legal, for enforcing the definitions, and for accepting additional definitions of legality.

e. There shall be a means to describe data, and there shall be a means to operate upon such descriptions. Descriptions of the data shall be distinguished from the data described.

These two requirements can be summarized as requiring "typing" or "strong typing" of the data. Most "state-of-the-art" procedural languages of the last decade provide for "strong typing," including Ada, but the idea of applying typing to the persistent data of a file system or data base is relatively new and requires justification.

As discussed above, a project will have a great number of data objects and operations on these objects. Typing reduces the number of different operations that can be applied to a given object and the number of different objects to which a given operation can apply. Without typing, the number of "meaningful" object/operation combinations is a very small subset of the total number of such combinations, in that a particular operation was intended to be applied only to a certain small number of objects and hence may cause unpredictable results, violations of preconditions, destruction of valuable information, and in general violations of engineering disciplines when applied to all others.

An important aspect of data management, which is more widely recognized as a crucial aspect of modern programming languages, is the separation of the structure and rules about data from the data itself. This concept is so widely accepted for programming languages that it is not normally felt necessary to justify it; however, some of the main reasons are rehearsed here.

Firstly, data is not normally operated on by only one user, but is operated on by many users. Making its structure and the rules about the data explicit mean that the several users have a single common understanding about the nature of the data.

Secondly, in any reasonable software project, there will be a large number of different kinds of data and a large number of specific operations. The great majority of the operations will "make sense" from a human viewpoint only when applied to a very small number of the kinds of data. It may not be unduly restrictive to specify that individual operations must apply to a single kind of data. Unfortunately, a user may, by mistake or with malice, request any operation on any piece of data. Thus typographical errors, misspellings, and other slips of the fingers or mind can result in requests to compile a tape drive or sort a progress report. An important goal of the facilities offered by the CAIS interface is to minimize the effects of human fallibility by refusing to perform operations that do not make sense. Specific requirements on enforcement of legality are considered later.

The definitions of what is legal in a system originate entirely with the people who design and build it; the system really only enforces their decisions. In most software systems, the rules are complex, contradictory, and largely unknown because they are created as by-products or unforeseen effects of decisions about efficiency, usability, or other concerns. It is a goal of the CAIS to make the rules governing the data and operations of a programming project as explicit and straight-forward as possible, while allowing them to be specified on an individual, company, or project level.

Thirdly, there is a need selectively to allow or prohibit certain operations on certain data requested by certain users or processes. Some specific examples of the kind of thing intended are given below.

f. There shall be a way to develop new data descriptions by inheriting (some of) the properties of existing data descriptions.

It is desirable to be able to derive new descriptions from existing ones. This results from the observation that there are natural ways in which some items of data are related to others and the conviction that the support mechanisms should conform to this natural "way of the world". While it is possible to develop such new descriptions independently of the existing ones, there are many advantages to providing support for inheritance of properties. If there is an orderly, supported means for deriving such related descriptions, the process will not be an ad hoc one prone to the errors and problems of ad hoc processes. The ability to inherit properties of the descriptions also will reduce the proliferation of independent relationships and properties. Perhaps most importantly, this capability will make it easier for users and projects to tailor the data collection to their own needs and to organize the structures of their data in natural ways.

One of the most important aspects of the inheritance of properties of data descriptions is that this allows tools which operate on an existing type to be used unchanged on new types which are descendants or derived from existing types. For example, we may have an editor which operates on a type "text". If we derive a new type "Ada-source" from text, then the editor should still work on this type, and correctly manipulate the attributes of this type, although the more specialized type may have additional attributes.

g. The relationships and properties of data shall be separate from the existence of the data instances.

h. The descriptions of data and the instances of data shall be separate from the tools that operate upon them.

The motivation behind these requirements is two-fold: (1) to establish a distinction between an item of data and the particular relationships in which it participates or the particular values of any of its properties and (2) to establish the assertion that the information and knowledge regarding these relationships and properties and their interpretation is controlled within the data collection and is not embedded in tools which are external to the collection.

The first part asserts that the identity and persistence of an item of data is distinguished from and potentially lasts longer in time than the particular values of any of its properties or relationships. That is, the values of the relationships and properties may change over time without changing the data item itself. This notion is to be distinguished from the assertion implied in *(e.)* that the descriptions to which the data items conform are separate from the existence of the data items themselves; both ideas are included here, and they are both important although independent of one another.

The second part asserts that all knowledge of the structure of the data items is retained as data within the data collection as opposed to any external tools. It may always be the case that tools may ascribe some additional meaning to a particular property or relationship, but the relationships and properties of general interest are not permitted to be defined strictly in tools which are external to the data collection. This constitutes a decision that all information about the data will be retained and controlled by the data collection.

3.2 Specifics of a Typed ERA Model

The previous section presented two concepts in isolation: the ERA model and the need for typing of data base objects. To combine the two, we recognize that the "objects" of the ERA model are the nodes, relationships, and attributes. We can now be somewhat less abstract in discussing what typing is and what advantages it brings.

(It was found to be very difficult to arrive at a definition of typing that satisfied more than a few people for more than a few days; the RAC takes the safe route of using a superficial definition and adding examples and further requirements.)

TYPING *An organization of entities, relationships, and attributes in which they are partitioned into sets, called entity types, relationship types, and attribute types, according to designated type definitions.*

Entities have a type; entity types might, for example, determine the types and number of each type of attributes and relationships required of an entity of that type.

Attributes have a type; attribute types might, for example, determine the form, format, number, and range of values required of attributes of that type.

Relationships have a type; relationship types might, for example, determine the types and number of the entities participating in relationships of that type and the types and number of each type of attribute required of a relationship of that type.

The CAISa designers have chosen the rather elegant approach of storing typing information as a node - relationship - attribute structure in the data structure it describes. For example, a node at a certain place in the structure defines a particular node type; the attributes that instances of that node type have are denoted by relationships to attribute type definition nodes. Likewise the relationships that nodes of that type may have are denoted by relationships to relationship type definition nodes. See Figure 4. Obviously there must be "built-in" definitions of the types used in the type definition structure.

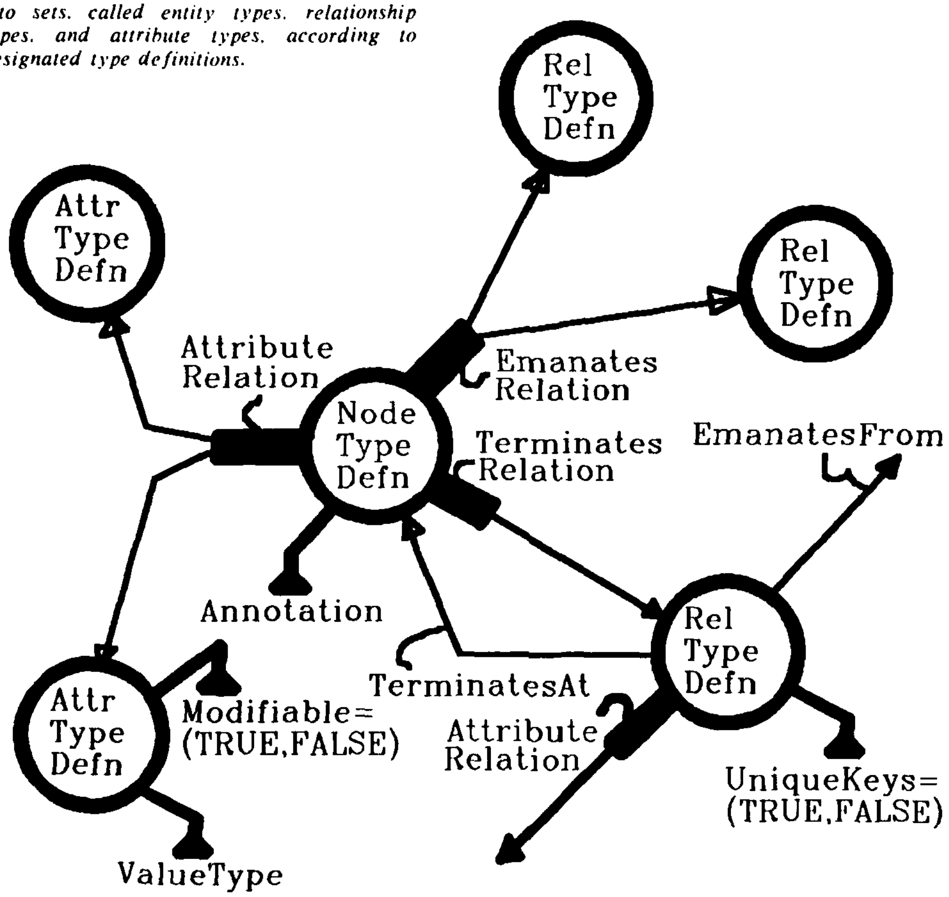


Figure 4 – Type Definition Structure

Attributes on the type definition nodes and relationships specify such things as initial values, cardinality (minimum and maximum allowable counts), sub-ranges, and accessibility. The structure is sufficiently rich -- and the resulting typing mechanism sufficiently powerful -- to allow project managers and individual programmers to tailor the environment completely to taste. (Note that a project manager may allow or prevent programmers from defining their own types by manipulating access controls on the definition structure.)

3.3 Types and Operations

The facilities provided by the CAIS shall enforce typing by providing that all operations conform to the type definitions.

The type of a thing cannot be discussed usefully in isolation from the functions that access the thing. These functions embody the type definition.

Typing establishes discrete sets of each of the nodes, relationships, and attributes of the data, probably a relatively small number of each, and explicitly associates a (also probably small) set of operations with each. These sets are called "types" and their members are "instances of that type." The effect is that the great mass of previously non-meaningful operation-object pairings now have a uniform result: rejection of the request as "illegal" with no change to any data. This has the following desirable effects:

- reduces (manages) the overall complexity of the system.
- reduces the potential for harm of inevitable human errors.
- allows "building in" of rules that in traditional systems were expressed in the code of tools and command procedures (and usually bypassable) or externally as guidelines or conventions.

"Typing" means that the database management software knows what kinds of nodes are created and manipulated by the tools, what relationships there are between nodes, and what attributes are associated with the nodes and relationships. Because the database management software has this understanding, it is able to perform many functions that are impossible in an untyped database like the node model of DOD-STD-1838, such as

- having more freedom to choose the physical representation of data in the database, while continuing to present data to the tools in the way the tools expect to see it;
- allowing different tools to see different aspects of the same data nodes. For example, project management tools might be interested in TEST_STATUS and AUTHOR attributes and relationships, while configuration management tools would be more interested in DERIVED_FROM relationships;
- altering the representation of data items to meet the needs of different processors.

Typing is also an important help with maintaining the integrity of the database. For example, if the database management software is told that every OBJECT_CODE node must be DERIVED_FROM a SOURCE_CODE node, it can enforce this. However, problems arise because the data objects are "persistent;" they continue to exist when the programs controlling them are not active. This distinguishes typing in the environment from typing in a programming language. Most of these problems have to do with the possibility of changes to type definitions while instances of those types exist. CAIS typing therefore cannot be identical to that of Ada, but should be "as close as possible" to minimize confusion (cognitive dissonance) of tool writers.

Every entity, relationship, and attribute shall have one and only one type.

At the time the RAC was written, two models were discussed: (1) every object has exactly one type and types are arranged in a directed graph or (2) objects could be of several types. We concluded that the requirements we wanted to express could be expressed in the two models in equivalent ways, but that it was easier to express them in the first model with this requirement (and later requirements that allow a given instance of a type to be operated upon as if it were actually an instance of another type). The CAIS designers were allowed to adopt either model and also chose the first.

It should be noticed that the RAC does not specify a particular failure mode for a type mismatch. Consider the following: a tool operates on nodes of a particular type; internally, the tool "opens" the node it is to work on and updates a particular attribute defined for the particular type. If the tool were given a node of an incompatible type, it could *fail* at any of *three times:* before the tool is invoked, at the time it "opens" the node, and at the time it access the attribute. Note that the third failure mode depends on a mismatch in *attribute* type, not node type.

3.4 Rules about Type Definitions

The CAIS type definitions shall

- *specify the entity types and relationship types to which each attribute type may apply.*
- *specify the type or types of entities that each relationship type may connect and the attribute types allowed for each relationship type.*
- *specify the set of allowable elementary values for each attribute type.*
- *specify the relationship types and attribute types for each entity type.*
- *permit relationship types that represent either functional mappings (one-to-one or many-to-one) or relational mappings (one-to-many or many-to-many).*
- *permit multiple distinct relationships among the same entities.*

- *impose a lattice structure on the types which includes inheritance of attributes, attribute value ranges (possibly restricted) relationships, and allowed operations.*

These requirements are basically the result of applying the typing requirements to the entity-relationship-attribute model; they establish type definition requirements for each of entities, relationships, and attributes. Their wording implies, but does not explicitly require, that attribute and relationship type definitions be independent of the entity type definitions that refer to them.

The requirement that two entities be able to be connected by more than one relationship is a practical one, based on such situations as a single user being the "owner," "author," and "last updater," of an Ada source entity.

The last bullet above is a major departure from the kind of typing found in the Ada language, requiring further discussion:

It is clear that: (1) we must have extensibility and (2) users and projects will want to particularize the data structures to their needs. Users need to be able to define new types that are extensions of existing types, such that operations and tools that expect entities of the old (base) type will also accept and work correctly on entities of the new type. The new type must therefore have the same attributes and relationships that the base does, plus any additional ones. Attribute types may have more restricted ranges in the new type than they did in the base, but obviously may not be less restricted or different in any other way. The tool or operation should not need any kind of recompilation or other preparation to operate on the new type. A type mechanism that works in this way is said to have a lattice structure. The authors of the RAC did not intend that the formal mathematical definition of the word "lattice" be used here; the CAIS designers need not follow the formal definition. A possible restatement of the last bullet in the RAC text above, to eliminate the word "lattice," would be

- permit the definition of new entity types whose definitions are derived from the definition of one or more existing entity types.

To give a concrete example, there might be an node type named "**ProgressReport**" that has a text attribute in a certain format, a set of attributes, and a set of relationships. See Figure 5. Some number of tools may

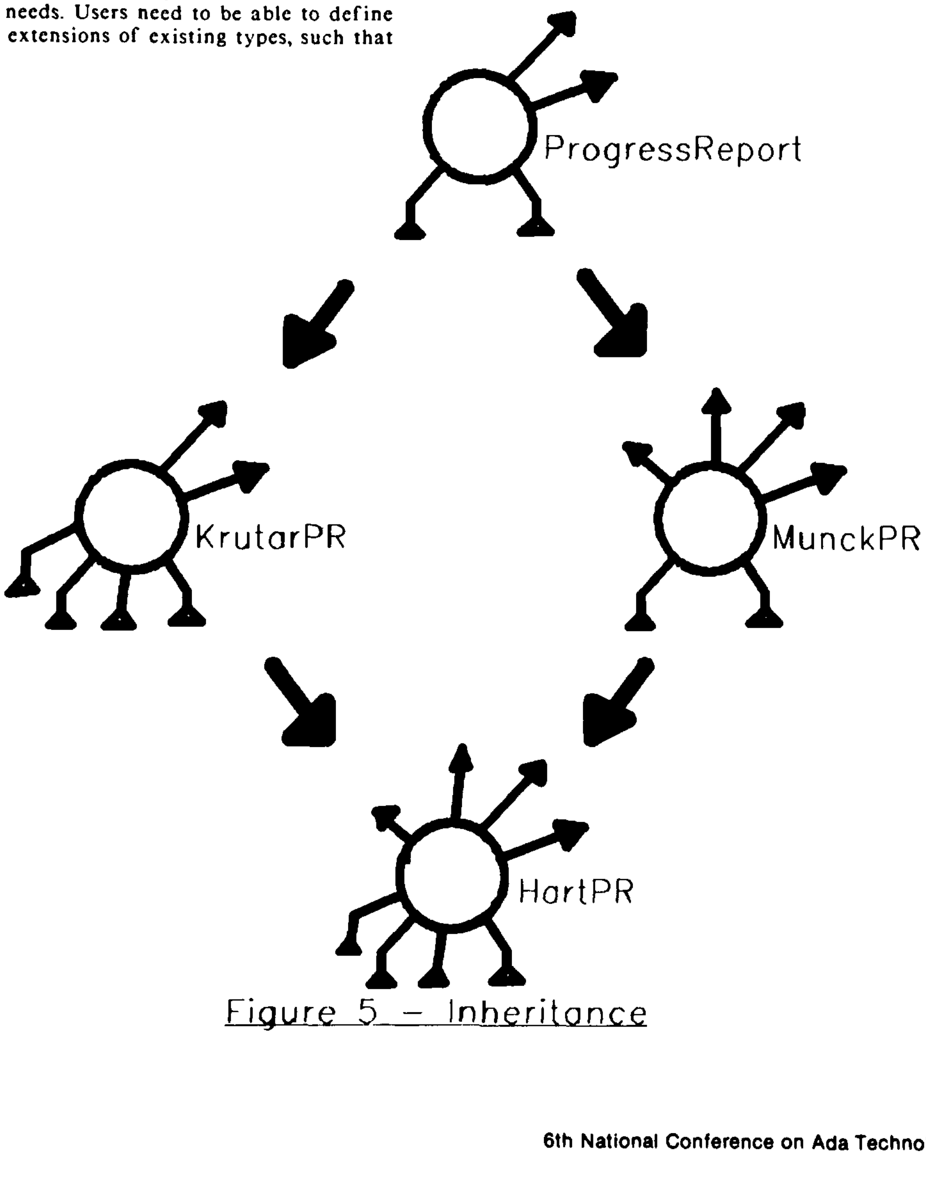


Figure 5 - Inheritance

exist that manipulate these objects to produce summaries, update Pert charts, and so forth. A manager named Krutar may want to add a couple of attributes for data that he's particularly interested in. He should be able to define a new type named **KrutarPR** such that objects of that type can still be processed by the existing tools and by new tools that need the new attributes. These new tools would not necessarily be able to process objects of type **ProgressReport**.

Another manager, Munck, might want his progress reports to include relationships to all program source and documentation mentioned in the text, so that he can look at it easily while reading the report. He could define a type named **MunckPR** that has those relationships.

Finally, because progress reports are generally kept for the life of a project and managers are not, the day may come when manager Hart must take over for Krutar and Munck. He may build new tools that operate on a progress report type that has both Krutar's additional attributes and Munck's additional relationships, called **HartPR**. Moreover, some of his tools may have to handle old objects of type **MunckPR** or **KrutarPR**, or even **ProgressReport**.

It need not necessarily be invisible to the tool that it is being asked to operate on an object of a different type than that for which it was intended; it could actively decide whether or not to operate on an object after asking the CAIS whether or not the object type is derived from the original type. For example, a tool written to use a particular attribute found in type **MunckPR** would ask if the object is derived from that type, indicating that the attribute is present. It would therefore operate on objects of type **MunckPR** and **HartPR**.

3.5 Type Definition

The CAIS shall provide facilities for defining new entity, relationship, and attribute types.

CAISa will include a number of pre-defined types: node types such as file, directory, process, user, and device; relationship types such as parent/child, owner of/owned by, members, and includes; and attribute types such as integer, float, string, and date. (Also "built-in" will be the components that make up the type definition structure.)

As environments evolve over time and new tools get installed, it must be expected that new entity, relationship, and attribute types need to be introduced. Consequently, the CAIS needs to provide facilities for defining such types. Moreover, such additions must be possible without impacting installed tools that are not logically affected by the change, i.e., there must not be a requirement to recompile such tools or a requirement to restructure the existing data base as a consequence of such additions.

The facilities provided by the CAIS must be such that there is no requirement for the tool writer to manipulate type definitions in terms of a single overall description of the data base. A tool writer ought to be able to concentrate on the subset of the typing information relevant for his or her work; moreover, security considerations may require that the tool writer not have access to information not needed in performing his or her tasks.

There should be no restrictions on the creation of new definitions, e.g., restrictions that could be caused by naming issues and not easily abided by the tool writer.

These objectives are motivated by the desire to have a large degree of flexibility in extending the type descriptions of the CAIS.

The CAISa designers have defined a **Specialization** relationship type in the type definition structure with the following meaning:

> If type definition node B has a **Specialization_Of** relationship to A, B is said to be a *Specialization* of A. This means that B has all of the same components as A but may place additional restrictions on those components and may have additional components.

A tool that works on instances of A will also work on instances of B because the components that it needs are available and it is unaware of B's extra components. However, it may fail in some cases if it violates one of B's extra restrictions, where the same action would not have been a violation for A.

The relationship **Specialization_Of** is bi-directional, with the reverse direction named **Generalization_Of**; in the above example, A is a *Generalization* of B. A given type definition may be a generalization of several other types and also a specialization of several different types.

In Figure 6, X, Y, and Z are specializations of W and Z is a specialization of X and Y. A tool that works on instances of W will also work on instances of X, Y, and Z.

3.6 Changing Type Definitions

The CAIS shall provide facilities for changing type definitions. These facilities must be controlled such that data integrity is maintained.

A software engineering environment must support the evolution of the kinds and organization of information that pertain to projects, in order to support the integration of ne·. tools and to accommodate changing project needs. The ability to change the type definitions of the data has a number of implications on the CAIS support for data. For example, one may want to delete, add or modify an entity attribute. Deleting an attribute in an entity type will result in the deletion of that attribute in all instances of that type. Similarly, adding an attribute will result in increasing the storage for instances of that type with the possibility of initializing the storage for that attribute. Modifying an attribute type (i.e., changing a range constraint) will require that all instances be checked to ensure that the values of these instances conform to these new constraints.

The ability to define new types through derivation is not sufficient to support the need to change type definitions in that one may want to modify existing types that are known to a set of tools without modifying the tools that use the type.

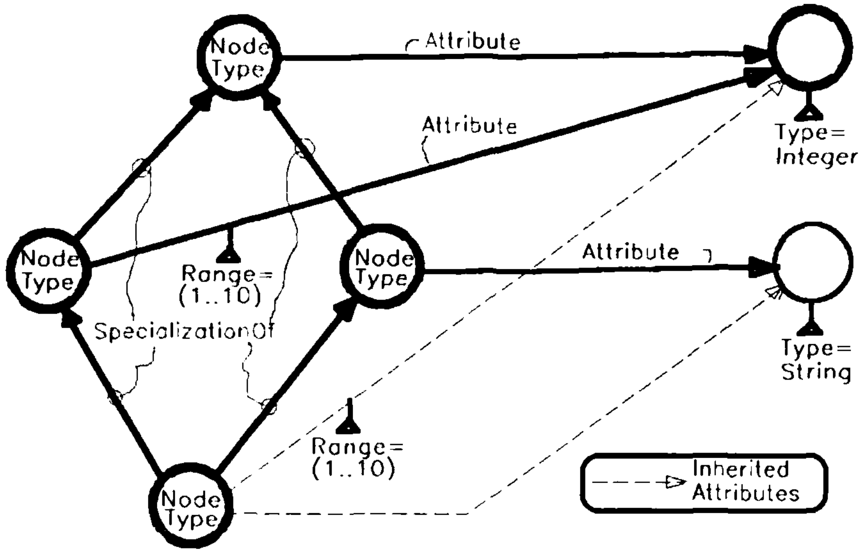


Figure 6 – Specializations

Type changes also include the ability to redefine the structural characteristics of the data. For example, one may want to change a relationship from being optional to mandatory, requiring the CAIS to ensure that all entities that can be related by this relationship in fact are related.

In practice, it is expected that types that have been in existence and use for awhile will not be changed; new needs will be met by creating specializations. Definitions will be *changed* mostly when the definer is correcting a mistake in the definition. However, the CAIS must still be able to handle instances of that definition created before the mistake was corrected.

4 Conclusions

The general node structure of DOD-STD-1838 provides an extremely expressive and flexible framework for storing and manipulating the data of a programming project; the structures found in older PSEs are generally small subsets of it. The proposed upgrade of 1838 to a *typed node structure* will add powerful facilities for insuring the integrity and correctness of the data. Making the type definitions explicit in the same data structure will support creation of some very powerful general purpose tools that use the definitions to guide their processing.

It is expected that early projects using a CAISa implementation will begin with a simple hierarchical structure like that of UNIX$^{(TM)}$ because of familiarity; the predefined types will include a simple superset of UNIX. As needs are recognized and handled, the structure will evolve to something much more complex and better suited to the needs of the company, project management, and individual programmers.

Implementations of DOD-STD-1838a will be at the leading edge of the state of the art in programming environments. It may someday be recognized as one of the most important stepping-stones in our path to mastery of the art/science of software.

5 Acknowledgments

The section of the RAC Rationale from which much of this material was taken was the result of many meetings and literally thousands of ARPANet messages; now that the spelling and punctuation has been uniformly "Americanized," it is impossible to determine the origin of any particular piece. Tim Lyons wrote much of the original text and did more than most to correct the author's (or "editor's") misunderstandings. Text and comments also came from Tricia Oberndorf, Frank Belz, Tony Gargaro, Hal Hart, Mike Horton, Jack Kramer, Rudy Krutar, Dit Morse, Erhard Ploederder, Ann Reedy, Andy Rudmik, and Edgar Sibley. Ultimately, though, many controversies were left unresolved and any remaining mis-statements are the fault of the author.

6 References

[CAIS86] *Military Standard Common APSE Interface Set*, United States Department of Defense, DOD-STD-1838, 9 Oct 1986

[CAIS87] *CAIS Reader's Guide for DOD-STD-1838*, Institute for Defense Analysis, 14 Aug 1987

[RAC86] *DoD Requirements and Design Criteria for the Common APSE Interface Set (CAIS)*, KAPSE Interface Team, Ada Joint Program Office, 4 Oct 1986

[RAC87] *Rationale for the DoD Requirements and Design Criteria for the Common APSE Interface Set (CAIS)*, KAPSE Interface Team, Ada Joint Program Office, 18 Nov 1987

[STONEMAN80] *DoD Requirements for Ada Programming Support Environments*, "STONEMAN," Feb 1980

1 Biography

Robert Munck received a degree in Applied Mathematics from Brown University and stayed on to teach programming languages and operating systems. He has worked at SofTech, where he was one of the developers of SADT, at Prime Computer, and currently is at MITRE. He has been active in the development of programming environments for twenty years. His current project is a prototype implementation of a multi-level secure DOD-STD-1838a on the Intel 80386 under AJPO sponsorship.

Ada Implementation in Management Information Systems

Major Terence Fong   Colonel Roy F. Busdiecker   Mr. Martin Johnson

U.S. Army Information Systems Software Center

ABSTRACT

In 1985, the Department of the Army directed that Ada be adopted as the single, common computer programming language for all systems. Despite initial estimates of high costs, high risks, and increased development times, the Army led the way within the Department of Defense in the use of Ada for information systems. This paper describes the progress of the Information Systems Software Center towards development of new systems utilizing the Ada language and identifies the challenges in achieving that goal.

INTRODUCTION

Several years ago, the Army began development of the Combat Service Support (CSS) Standard Army Management Information Systems (STAMIS), to support the critical functional areas of personnel, force accounting, supply, transportation, maintenance, ammunition, finance, and medical. STAMIS are installed on various systems within the Sustaining Base and Theater/tactical environment.

The mission for development, integration, testing, extension and maintenance of STAMIS is the responsibility of the U. S. Army Information Systems Software Center and its five geographically dispersed software development centers. The integration of the Ada programming language has become a key element in the STAMIS modernization plan. Within the Information Systems Software Center a management level Ada Implementation Group coordinates transition to and implementation of the Ada language.

ACCOMPLISHMENTS

Our primary goal is the development of all new systems as well as those existing applications requiring 25% or more revisions in Ada. In order to do so, it was necessary to acquire Ada compilers, tools, and environments for training and development and to obtain hands on knowledge of Ada software engineering through participation in other Ada programs and Ada Beta tests.

Thus far, six STAMIS are scheduled to be developed using Ada. These systems are the Standard Installation/Division Personnel System-3 (SIDPERS-3), Standard Army Finance System - Redesign (STANFINS-R), Standard Army Retail Supply System-Level 2AC/2B (SARSS-2AC/2B), Department of the Army Movements Management System-Redesign Phase II (DAMMS-R Phase II), Standard Property Book System-Redesign (SPBS-R), and the Combat Service Support Command and Control System (CSSCS).

SIDPERS-3 is currently under development at Software Development Center, Washington. It is a redesign of the Army's automated personnel system which will replace the three separate field personnel systems now in use (Active Army, Army Reserve and National Guard) with a single system. SIDPERS-3 will fulfill the personnel accounting and reporting needs in the peacetime, mobilization, and wartime environments.

STANFINS-R is a new, automated financial system being developed to replace several existing Army accounting systems. STANFINS-R consists of seven modules implemented in two subsystems. Subsystem I contains the commercial accounts, travel I and disbursing modules. Subsystem II contains the cost accounting, general accounting, performance measurement, program budget and input control modules.

SARSS-2AC/2B is part of a family of SARSS systems which will operate at the unit level, Division and Corps. SARSS 2AC will perform time sensitive supply actions, such as lateral referral of requisitions, routing of unfilled requisitions, and disposition of items declared excess. SARSS 2B will perform less time-sensitive management functions such as demand history, demand analysis, levels computation, document history maintenance and inquiries.

DAMMS-R Phase II is the second phase of a two phase implementation of a movements management system. It is designed to meet the basic transportation automation needs in a theater of operations down to Division level. *DAMMS-R consists* of several modules: cargo movements; transportation movements address; container and freight; recoded from the Phase I implementation, plus the added functions of unit movements, mode operations, highway regulation and inter theater; all coded in Ada.

SPBS-R provides standard, automated functional procedures and processes for property accounting, equipment management, and asset reporting in both Divisional and non-Divisional units.

The CSSCS is the CSS node of the Army Tactical Command and Control System. It will automate the collection, analysis, and distribution of key elements of information from logistical, personnel and medical functional systems for use by the battlefield commander. The CSSCS will be the key decision support system in the CSS battlefield area.

To support these efforts, the Information Systems Software Center has acquired the system software necessary to train our programmers and develop applications at each of our software development centers.

The Information Systems Software Center acquired the Intermetrics Multiple Virtual Storage (MVS) Ada Compiler version 5.5 and the Meridian AdaVantage MS-DOS Ada Compiler Version 2.0A and Ada Workstation Environment. The Intermetrics MVS Ada Compiler has been installed at Software Development Centers, Washington and Ben Harrison, to train systems programmers and application developers and for development of systems targeted for large computers running MVS. The Meridian AdaVantage Ada compiler and Ada Workstation Environment were purchased on a indefinite quantity, indefinite delivery contract for use throughout the Army. Initial copies have been installed at each of our software development centers and at the United States Military Academy for integration into the course of instruction there.

We also installed two Rational Ada development environments at Software Development Center, Ben Harrison. These systems will be used to develop STANFINS-R. Additional sophisticated Ada development systems are being acquired for each of the other software development centers.

Additionally, fourteen Sperry 5000/80s with the Telesoft TELEGEN II Ada compiler and Ada programming support tools have been purchased to support application development and training requirements.

To gain more knowledge of Ada software the Information Systems Software Center is participating in the WWMMCS Information Systems (WIS) Joint Program Management Office (JPMO) Software Development Maintenance Environment (SDME) Working Prototype System beta test. The SDME consists of Ada tools integrated into an environment which supports development of Ada software. The tools include an editor, a pretty printer, standards checker, document formatter, configuration management/project management tools, Ada program metric functions, and a reusable Ada software cataloging facility. These tools are used with the host Ada compiler running on a DEC Micro VAX II.

To support fielding of the STAMIS, the Command acquired the XDB Data Base Management System (DBMS). XDB is a relational DBMS with an SQL capability. It is the common DBMS for STAMIS development and fielding and will operate on UNIX, MS-DOS and XENIX based minicomputer and microcomputer systems. XDB will also provide the Department of the Army approved standard Ada SQL binding later this year.

Critical to our primary goal is the need for a standard development methodology. To analyze this problem, a "Skunkworks" team was established. The objectives of this group are:

- To define the Information Systems Software Life Cycle Model and the methodology through which Army management information systems are to be developed.

- To define an immediate short-term solution.

- To define the mid and long term solution.

- To define an overall implementation plan.

- To extend the methodology Army wide.

The short term objectives will establish a foundation for software engineering within the Command. Mid term objectives will work towards completing the definition of the command software development engineering methodology and will provide the bridge to long term objectives of incorporating both information engineering and software engineering, and enhancement by organizational/corporate project management systems.

ISSUES

Despite these accomplishments, several technology gaps are hindering our efforts to field STAMIS in Ada. These technology issues include adoption of a single, efficient Ada SQL binding and a standard POSIX Ada language binding, Ada programming support tools for microcomputers, minicomputers and large computers, application generators for Ada code using the technology of 4th Generation Languages, (4GL) software portability/reusability development guides and large scale transaction processing support. The Ada Joint Program Office (AJPO) and the Software Engineering Institute Office are working towards identification of an Ada SQL binding. At issue is a solution which precludes the embedding of a foreign syntax into the Ada program. The solution must take full advantage of Ada's strong data typing capabilities and allow the use of Ada sensitive tools for debugging the application. The Army is moving towards the open systems architecture currently being defined by a group of national and international standards efforts. The Army plans on adopting the Portable Operating System Interface Environment (POSIX) as a common specification for UNIX-like operating systems when it becomes a Federal Information Processing Standard. Critical to this standard is the Ada language binding to POSIX. The Information System Software Center is actively participating in the P1003.5 POSIX Ada Language binding standardization effort; however, a full use standard is not expected to be approved for two years. Another issue is the need for Ada environment support tools for microcomputer, minicomputer and large computers. Ada compilers by themselves are not enough to support our software engineering requirements. Our next initiative will be to acquire additional Ada programming tools for our development systems. We hope that the SDME will provide the foundation for the development of a mature Ada Programming Support Environment. The Information Systems Software Center is also investigating the feasibility of developing an application development tool which takes full advantage of the high productivity 4GL technology and the portability/reusability of Ada to generate Ada applications. We are encouraged by the information that has been collected from vendors and are optimistic that 4GL technology can be transferred to Ada. Another area of concern is a sophisticated Ada source code library management tool which will make reuse of Ada code easier. As the number of Army Ada applications grow, Ada software reusability tools will become increasingly important. Finally, large scale transaction processing support will be a necessity for Army information systems using Ada. Without some type of teleprocessing monitor, efficiency is likely to be poor and uneconomical in comparison with other programming languages; such as COBOL. Concentrated research and development in this area is needed to overcome this deficiency.

CONCLUSION

The Department of the Army is committed to implementing the Ada programming language for information systems. Our accomplishments are growing each day, but the technology issues identified are hindering our efforts and they must be resolved soon in order for us to continue. To be successful, the Department of the Army, academia and commercial industry must combine their resources to further the use of Ada in all systems.

Colonel Roy F. Busdiecker is the Commander of the U. S. Army Information Systems Software Center, Fort Belvoir, Virginia where he is responsible for the Army's central software development and maintenance activities. His military career includes over 25 years in the technical management of computers, communications, command and control systems and electronics. COL Busdiecker has a BS from the United States Military Academy and MS in Electrical Engineering from Stanford University.

Major Terence Fong is the Executive Officer, U. S. Army Information Systems Software Center at Fort Belvoir, Virginia. He was previously an ADP Officer in Executive Software at the Computer Systems Support Center. MAJ Fong is a member of the IEEE, Computer Society, and SIGAda. He is currently the Chairperson of the P1003.5 POSIX Ada Language Binding Working Group. MAJ Fong received a BS from the United States Military Academy and an MS in Management Information Systems from The George Washington University.

PICTURE NOT AVAILABLE

Mr. Martin Johnson is a Computer Systems Programmer, U.S. Army Information Systems Software Center, Fort Belvoir, Virginia. He has held a number of information management positions with the Army during the past ten years. Mr. Johnson has a BS from the University of Wisconsin-Oshkosh, and an MS in Human Resource Management from the University of Utah.

Commander, U.S. Army Information
Systems Software Center
ATTN: ASBI-CO (STOP C-5)
Fort Belvoir, VA 22060-5456

BENEFITS REALIZED FROM USING ADA, MODERN SOFTWARE ENGINEERING PRACTICES, AND ADVANCED ENVIRONMENTS FOR DEVELOPING LARGE, COMPLEX SOFTWARE SYSTEMS

by Robert T. Bond, *Vice-President, Marketing*

Rational
1501 Salado Drive
Mountain View, California 94043

ABSTRACT

Improvements in the development of complex software applications are being brought about through the use of Ada®, modern software engineering practices, and a'-vanced programming environments. This paper examines the need for such improvements and discusses what is available to facilitate software development today. The case study describes one company's experiences in confronting the need to increase dramatically their software engineers' productivity.

SOFTWARE DEVELOPMENT CHALLENGE

Systems in which software is a major component are among the most intellectually complex human endeavors. Unlike other artifacts, however, software is an intangible medium. Its characteristics cannot be measured precisely, short of counting the lines of code in a given program or producing a relative complexity measure. Software development involves the creation of many more products than just source code itself. Therefore, for anything beyond a small system, the effort of a team of developers is required; this human factor further complicates the problem. Given the ever-growing capabilities of our hardware, along with an increasing social awareness of the utility of computers, there is great pressure to automate more and more applications of increasing complexity.

Indeed, the demand for software far exceeds our ability to supply it. As a recent study by the U.S. Department of Defense suggests, the demand for software is increasing at a yearly rate of about 12%, while our net productivity is increasing at a rate of only 4% per year. Software development is still an extremely labor-intensive process, and yet the programmer labor force is growing at only 4% per year.

The continuing mismatch of software supply and demand results in higher costs for developing software and maintaining it over its lifetime. However, high software costs reflect only a part of the problem; perhaps more important is the quality of software produced. All too often, projects exceed their budgets and schedules or fail to meet their expected requirements.

With software increasingly becoming an important element in the functioning of all complex systems, the major challenge facing the world economy today, including the aerospace and defense sector, is the production of high-quality and economical software, given the constraints of scarce human resources. Fortunately, three elements have emerged over the last several years that together provide a means to control the cost, reduce the risk, and manage the complexity of developing large, complex, software-intensive systems: sound software engineering principles, powerful programming languages, and advanced software development environments.

Software Engineering Principles

Simply stated, software engineering is the disciplined application of sound principles, such as abstraction, modularity, strong typing, and information hiding, that guide in the construction of complex software systems. Civil engineering has had many centuries of growth, during which time —through trial and error and more controlled experimentation—a set of accepted practices emerged to guide in the construction of dams, roadways, bridges, and other public works.

There is a far shorter history in the computer sciences than in civil engineering, but experience in developing many large, complex systems during the 1970s has led to a generally well-accepted and well-founded set of disciplined methods, such as structured design and object-oriented design. Such methods help to manage the complexity of software development by offering a fairly rigorous process of software development. Related approaches such as rapid prototyping also help to reduce risk by increasing the visibility of critical design decisions early in the lifecycle, thereby permitting early feedback to users, freedom to experiment with alternative approaches, and flexibility to make fundamental architectural changes with minimal impact on the fabric of the overall design.

High-Level Programming Languages

Closely related to advances in software engineering is the similar progress made in the design of powerful programming languages. Such languages are more than a vehicle for expressing implementations; they are a means of expressing and enforcing our design decisions. This is especially true of languages such as Ada.

Ada is best viewed not just as another programming language but as a vehicle for the application of modern software engineering principles. Furthermore, Ada has been developed with the issues of programming-in-the-large in mind. With constructs such as packages, generic units, and tasks, the developer has at his or her disposal a rich set of building blocks with which to construct large software systems. Using Ada as a design language as well as an implementation language provides a means of capturing design decisions in a fairly rigorous way; tools then can be applied to enforce these design decisions.

Advanced Development Environments

The third element that has emerged over the last several years to improve the productivity of software engineers is advanced software development environments. In the CAD/CAM industry, for example, there is a growing reliance on the use of powerful integrated and interactive tools. A decade ago, common practice was to do gate-level design with only paper and pencil, committing those designs to a breadboard to validate them. To do VLSI design today without tools would be unthinkable. Using VLSI tools, a mathematical representation of the design can be built and then manipulated and analyzed in a variety of sophisticated ways.

The software industry has witnessed the beginning of a similar trend. In the past few years, software development proceeded with only the most minimal bag of tools: compilers, editors, debuggers, and so on. More recently, environments such as the integrated, interactive Rational Environment™ have been developed, giving a much more seamless approach in providing a software engineering environment. In the Rational Environment, for example, all Ada programs are retained in the more semantically rich intermediate form, DIANA (Descriptive Intermediate Attributed Notation for Ada). All environment tools, much as in interactive CAD/CAM systems, work from this representation. For programming-in-the-small, this means that tools are much more integrated, thus simplifying the activities of the developer. Given a powerful representation such as DIANA, together with hardware acceleration, incremental compilation is possible, thus facilitating methods such as rapid prototyping.

Of course, software development involves much more than the activities of a single developer. For programming-in-the-large, there must be tools for configuration management and version control. There also must be mechanisms, such as Rational Subsystems™, for managing multiple simultaneous releases of a system without incurring excessive recompilation overhead. Especially for the domain of embedded systems, there must be host/target development environments, in which powerful tools are placed on a much larger host machine, developing code targeted for a possibly bare target machine.

The application of these three elements leads to increased productivity, which in turn controls the cost of development. Perhaps more importantly, risk is greatly reduced through the accelerated detection of errors and the automation of design checking, which otherwise would require the use of scarce and error-prone human resources. In addition, these three elements help to manage the complexity of developing a large, complex, software-intensive application. It is a fact that software systems are growing in complexity, not decreasing. It is not possible to reduce this complexity, but with modern software engineering practices, Ada, and an advanced programming environment, it is possible to manage the complexity.

A CASE STUDY

This case study shows how one company controlled cost, reduced risk, and managed complexity by adopting a new technology when faced with three new software development projects within the same time frame. Philips Elektronikindustrier AB (PEAB), Sweden's leading developer of command and control systems, both internationally and for the Swedish defense services, won a contract to deliver one of the largest software development projects ever undertaken: the 9LV Mk3 system, an integrated C3 and weapon control system for three new classes of warships. The three new classes of warships are the Swedish Goteborg-class coastal corvette, the Danish Standard Flex 300 multirole ship, and the Finnish Fast Patrol Boat.

Because PEAB had extensive experience in developing command and control systems, they were well aware of the limitations of their current technology. They needed to find a technology that was robust enough to apply to the development of the 9LV Mk3 project. In addition, PEAB wanted the 9LV Mk3 system to be the basis for a common architecture that would have a very long lifecycle.

PEAB had experience developing many fire control systems in assembly language and high-level languages, such as RTL/2 and Pascal. Over a 15-year period, they had developed approximately 25 different systems that ranged in size from 30,000 to 100,000 source lines of code (sloc). The system that PEAB had developed most recently, over a period of seven years, was a C3I system of approximately 700,000 sloc that required 300 engineering-years for completion.

Through their extensive development experiences, PEAB learned that certain characteristics appeared in all their systems development. First, requirements always changed during the project lifecycle; second, there were always many releases of the software; and third, there were close links between software development and software management. These characteristics, along with the critical shortage of software engineers in Sweden, led PEAB to determine that to be competitive on future large software programs, they had to realize a significant improvement in software development productivity.

PEAB began a process of evaluating new technologies. They wanted to evaluate the technology by applying it to an actual project that could be completed before the 9LV Mk3 project had to begin. After preliminary evaluations of many technologies, PEAB decided to evaluate Ada and the Rational Environment on a project. The Rational Environment is a comprehensive, lifecycle-oriented environment integrated around the Ada language. The objective of the Rational Environment is to provide a completely integrated set of capabilities that support and encourage the use of good software engineering practice through powerful, interactive tools.

PEAB chose the UndC project, a small but significant mobile Army command and control system, for evaluating the new technology. The UndC application involves the integration of high-resolution terrain maps stored on laser disk with computer-generated trajectory information. The

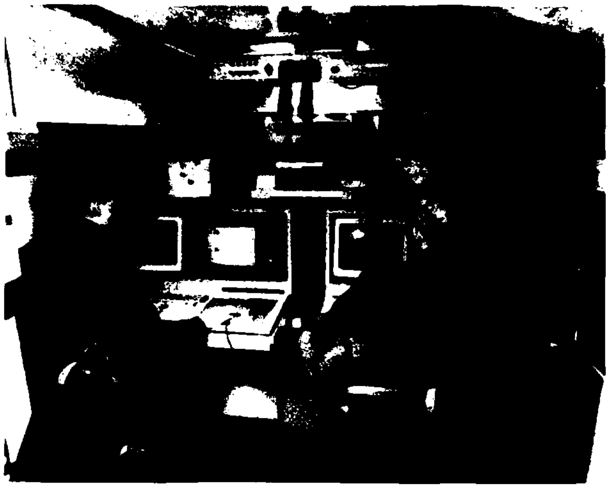
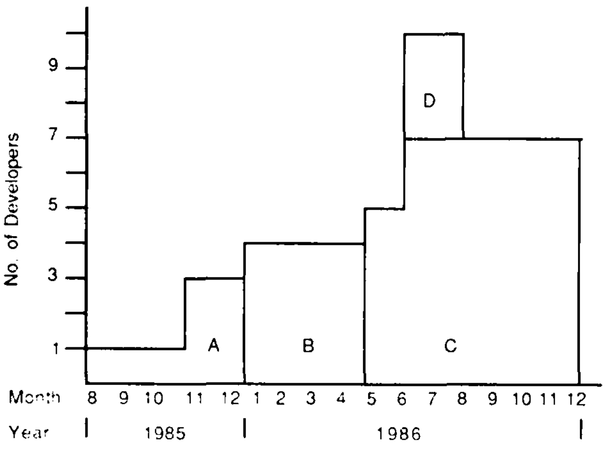


Figure 1. Van Used for Mobile Army Command and Control System

application is targeted to a DEC™ MicroVAX™ computer system mounted in a van, as shown in Figure 1.

The UndC project was PEAB's first Ada project with a delivered product. It is written entirely in Ada. PEAB's software engineers had no previous Ada experience before undertaking this evaluation project.

The project began in August 1985 and was completed in December 1986. The delivered code totaled 55,000 source lines of Ada code. The delivered product included user's manuals, internal documentation, and a project standards handbook. Figure 2 shows the project staffing during the development cycle.



A Software requirements analysis
B Design
C Implementation (coding, unit test, integration, system test)
D Three developers from another project joined the team to evaluate Ada and the Rational Environment. Although they contributed to the overall estimate, their primary interest was in evaluation.

Figure 2. Philips Project Staffing

The project was originally estimated to be 18,000 source lines of code. The estimates, based on assumptions of productivity using Ada and a conventional batch Ada compiler, were for 74 engineering-months and a project completion date of December 1986. Given a project start date of August 1985, this would have resulted in a productivity rate of 243 source lines of code per engineering-month.

The project expanded primarily because requirements were added and because PEAB unexpectedly had to develop a graphics subsystem. (The off-the-shelf graphics subsystem they had planned to use became unavailable.) Also, the project team took advantage of Ada's exception-handling capabilities to produce a very comprehensive error-handling system, which added to the size of the system but has already paid off in reduced maintenance costs.

The project moved to the Rational Environment in May 1986 at the start of the greatest resource consumption, the implementation phase. Because of the increased productivity realized by using the Environment, the project was completed on time, with only a minor addition of resources —six engineering-months. (Actually this addition represented three developers from another project who wanted to evaluate Ada and the Rational Environment for a large proposal effort.)

The cost of the delivered system, if projected productivity had been achieved, was $2.16 million. The actual cost of the delivered system was $0.76 million. The cost savings was $1.4 million. The acquisition cost of the R1000® was $690,000, and assuming a 48-month amortization period with zero residual value, PEAB's payback on their capital investment was 14X.

Technology Evaluation Conclusions

PEAB was able to control costs by adopting a new technology that was built on modern software engineering practice. They were also able to reduce risk; the UndC project was completed on schedule with basically no increase in staffing. Applying better software development technology significantly reduced the risk of this large program, even in the face of expanded requirements.

Many qualitative conclusions were drawn. The team of software engineers felt that they were better able to respond to changing requirements. The recruiting of software engineers became easier. Currently, 25–30% of the personnel for the 9LV Mk3 project are new employees of PEAB. The team also felt that the quality of the completed system was improved over other systems that they had developed in the past. The team further concluded that Ada supports modern design methodologies very well.

As a result of the technology evaluation project, PEAB determined that the technology was robust enough to be used on the large-scale, complex 9LV Mk3 program. PEAB then renegotiated the contract with their 9LV Mk3 system customers to use Ada and the Rational Environment on the project. Although Ada was originally specified for use on the system, both bidders for the contract had obtained waivers; Ada had not been bid on the contract originally. The contract had been won on the basis of using an RTL/2 environment.

The main factor behind the Ada decision was the desire to raise programmer productivity considerably, partly through the power of the language but also because of the quality of the available development environment, the Rational Environment.*

The 9LV Mk3 System

The 9LV Mk3 system is a new family of shipborne weapon control systems. PEAB is under contract to deliver the electronics suites for three new classes of warships that are the first systems to be built in this new family. These three new classes of warships all carry considerable firepower and can handle complex missions. As a consequence, the electronics systems must cope with the complexity of weapon systems typical for much larger ships while still making it possible for a limited crew to control the operation of the system.

The initial operating capability for the 9LV Mk3 system is more than 1 million source lines of code. The operational software runs in an environment of multiple Motorola® 68020 processors. The complexity of the 9LV Mk3 system comes from its requirement that the ships be multi-configurable, depending on operational needs. The system structure must have the necessary properties of being adaptable to changing demands from current customers as well as the customers for the new family of weapon control systems.*

Status of the 9LV Mk3 Project

Starting with no Ada experience at the beginning of the evaluation process, there are now more than 100 software engineers working with Ada, the Rational Environment, and object-oriented design. The detail design phase of the project is complete. The project is currently in the coding and unit-test phase. More than 100,000 source lines of code have been developed. The productivity currently being achieved on the project is consistent with the earlier evaluation project.

CONCLUSIONS

During the past few years, the emergence of software engineering practices has provided a means to control the cost, reduce the risk, and manage the complexity of developing software. Software engineering provides methodologies for large-system decomposition and abstraction. Ada facilitates the use of software engineering by providing a structure that directly captures the design of a system and automates the control of component interfaces. Software development environments are now available to provide automated support for the software engineering process.

PEAB's experiences show that the use of modern software engineering practices, an advanced software development environment, and Ada can have a significant impact on the economics of medium- to large-scale projects. They have made a long-term commitment to Ada and believe that the language is receiving long-term support from industry. From the results of their technology evaluation project and the productivity currently being achieved on the 9LV Mk3 project today, PEAB is confident that, with the use of modern software engineering, Ada, and the Rational Environment, they will be able to control the cost, reduce the risk, and manage the complexity of the 9LV Mk3 project.

Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

DEC and MicroVAX are trademarks of Digital Equipment Corporation.

Rational and R1000 are registered trademarks and Rational Environment and Rational Subsystems are trademarks of Rational.

Motorola is a registered trademark of Motorola, Inc.

Robert T. Bond has been vice-president of marketing for Rational since joining the company in 1983. Bob is responsible for all aspects of national and international sales and marketing at Rational.

Bob's background spans more than 16 years at Hewlett-Packard Company. He was responsible for forming the Application Marketing Division for HP, which grew, under his direction as Division General Manager, to $50 million in annual revenues and 15 application centers employing more than 400 people around the world. Bob was responsible for all aspects of the Division, including profit, growth, productivity, and customer satisfaction.

Bob received a BS degree in Engineering at Case Institute of Technology in Cleveland, Ohio.

* "The Implementation of Ada in a Large Shipborne Weapon Control System," paper by Roland Fors and Ulf Olsson.

SUPPORT ENVIRONMENT CONCEPTS FOR COST-EFFECTIVE TRANSITION TO ADA™ TECHNOLOGY

Peter Lempp

SPS Software Products & Services, Inc.
New York, New York

Abstract: Before modern software engineering principles and Ada™ came into common use, a huge amount of software was written in high-order programming languages like FORTRAN, COBOL, CMS-2, etc. The challenge now is to preserve the valuable parts of these software assets, and the application experience of software designers, when migrating to Ada. This transition is eased significantly if the development was done independently of a specific target language, designing with the assistance of a development support environment. From the resulting design, Ada source code can be automatically generated.

For software developed without a powerful support environment, one approach to migration is "Reverse Engineering," a process which recaptures design abstractions from existing software source code, then regenerates them in Ada. Though not fully automatable, this process shows advantages compared with direct language translators, since the resulting software incorporates the design and requirements specification as well, and is therefore easier to maintain and reuse. This paper discusses the methods developed, and experiences gained, in recently performed Reverse Engineering projects. It also describes the environment support needed to achieve the high degree of automation necessary for a cost-effective transition.

1. Background

Most companies and agencies which have developed application-specific software representing a large investment, and including a great deal of company know-how in design structures, algorithms and heuristics, now face an enormous problem: This software is written in languages like FORTRAN, HAL/S, CMS-2 or COBOL, and in many cases has lost its clear internal structure; often, too, it is not properly documented, or at least the existing documentation is no longer up to date.

In order to make a transition to Ada technology without losing the application-specific knowledge, more than a mere translation of existing source code to Ada code is necessary. This direct transformation from one language to another has been shown to be almost fully automatable for various programming languages /Sant86/, /HKPT87/. However, the resulting Ada source code from, e.g., a FORTRAN program is more "AdaTRAN" than a full use of the powerful language capabilities of Ada. What is worse, the transformed Ada program tends to be even less maintainable than the original code in the conventional programming language. Since there is no visibility of the underlying design structure, efficient maintenance, restructuring or other improvement of the resulting Ada software is extremely difficult.

These observations lead to the development of an approach which involves a higher degree of human interaction and more sophisticated tool support, but which in the long run provides a real unraveling of the system to its inherent higher level abstractions; it therefore provides significant potential to incorporate the existing software in the reusable Ada software libraries for the future (e.g., /BABK87/, /McNi86/). The approach is to undertake a Reverse Engineering of the software first, then to regenerate Ada source code from the recaptured design specification.

2. Software Reverse Engineering

2.1 Reverse Engineering Methodology

Reverse Engineering is the process of unraveling the system and software to its earlier life cycle development representations. During the Reverse Engineering process described here, one goes beyond the concepts discussed in, e.g., /ABCC87/, and reconstructs higher levels of abstraction, especially on the design specification level. One normally starts not with the source code alone, but also takes existing documentation, e.g., informal high-level requirements specifications, and especially input/output descriptions, etc., into account.

Figure 1 depicts the task graphically, and shows on the right-hand side the ultimate results of the Reverse Engineering process when performed with a comprehensive Computer-Aided Software Engineering (CASE) environment: requirements specifications, design specifications, (re-)generated source code in the appropriate programming language(s), and full traceability between the different representations. These representations are then suited for reliable maintenance and potential reuse of parts of the software package.

*Ada is a registered trademark of the U.S. Government (AJPO).

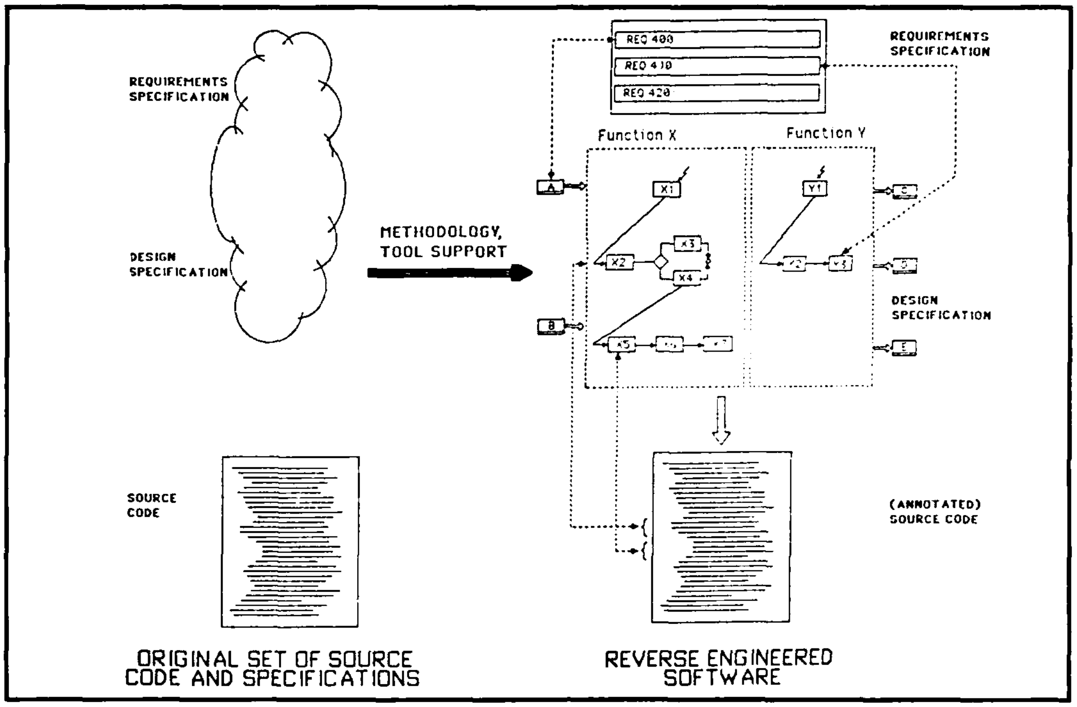

REQUIREMENTS SPECIFICATION
DESIGN SPECIFICATION
METHODOLOGY, TOOL SUPPORT
SOURCE CODE
REQ 400
REQ 410
REQ 420
REQUIREMENTS SPECIFICATION
Function X
Function Y
A
B
X1
X2
X3
X4
X5
Y1
C
D
E
DESIGN SPECIFICATION
(ANNOTATED) SOURCE CODE
ORIGINAL SET OF SOURCE CODE AND SPECIFICATIONS
REVERSE ENGINEERED SOFTWARE

Fig. 1: Reverse Engineering of existing software [here depicted in the notation used in the CASE environment EPOS (/LaLe86/)].

The overall procedure to (re-)establish requirement and design specifications can be broken down into the following fundamental activities:

(1) Formal source code analysis

(2) Functionality determination

(3) Establishment of interrelated specifications; and

(4) Verification

The two further steps necessary for a successful transition to Ada, namely,

(5) Restructuring of the reverse engineered design (if necessary), and

(6) Generation of Ada code from the design

are not directly part of the Reverse Engineering process, and will be discussed later.

The first activity in the Reverse Engineering process, source code analysis, captures all basic structures, such as hierarchical calling structure, "module" interfaces, internal/external data and its principal structure, controlflow and dataflow, as well as low-level input/output functions and interfaces to language-dependent library routines. The further breakdown of functions into their main blocks, relying not only on the subprograms/functions involved, but also on package-specific block separators like

- comment lines of a specific pattern,
- input/output statements, and
- specific types of labels, etc.

results in a formally structured design as shown in the example in Figure 2.

Whereas the first activity of the Reverse Engineering process is conducted solely on the formal parts of the software source code, the functionality determination is based on a broader spectrum of information sources. In general, these sources can be grouped into:

- the formal elements of the source code or their equivalent low-level design representation as a result of the formal source code analysis;
- internal documentation (e.g., comments or names/-labels) within the source code;
- "external" documentation (such as design specifications, requirements documents, input/output definitions, etc.).

The expected accuracy of additional information sources may vary. If there is a contradiction between conclusions drawn from the (formal) source code and those drawn from other documentation, the source code is taken as the authority.

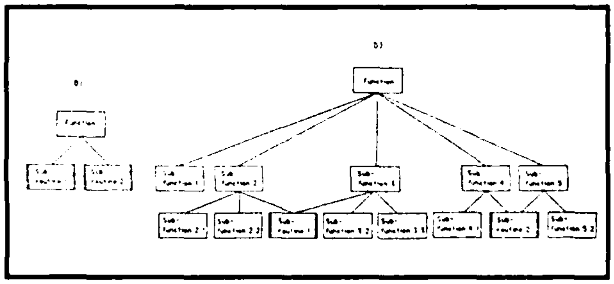


Fig. 2: Example of (a) calling structure and (b) respecified design hierarchy structure

Different strategies are possible to penetrate a formally restructured software design to recapture its "meaning," including various starting points like identified library routines, input/output processing modules or identified "selector modules," (i.e., modules that invoke different modules depending on specific internal states). Whereas the exact implementation of the strategy varies from one software package to the next, the general procedure of establishing a hypothesis of functionality which proves to be correct or leads to inconsistencies is the same. Hypotheses are drawn either from existing high-level documentation (top-down), or various indicators in the source code (bottom-up). Naming schemes, the knowledge of expected behavior from the application domain, or the deduction of functionality from known internal/external characteristics are helpful in this task. Figure 3 shows, e.g., a typical function in the avionics domain which can be deducted from its input/output characteristics.

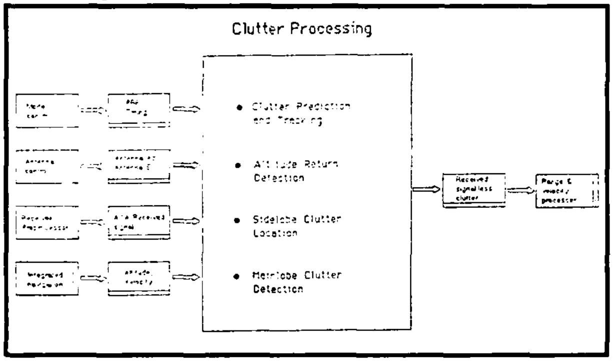

Clutter Processing
Clutter Prediction and Tracking
Altitude Return Detection
Sidelobe Clutter Location
Mainlobe Clutter Detection
Received signal less clutter
Range & velocity processor

Fig. 3: Example of input/output characteristics and components of a typical avionics subfunction

The process of <u>building consistent and interrelated specifications</u> from the existing requirements documents and the recaptured design information aims at establishing (formal) links between the different descriptions available as project/product documentation. Further, it aims at updating the specifications so that they reflect the current status of the software represented by the source code. The process of finding interrelated descriptions relies strongly on searching for equal or similar specification/naming patterns.

A <u>verification</u> of the results of the Reverse Engineering process is essential. After assuring completeness and consistency of the representations themselves, source code in the original programming language is regenerated from the design specification and compared for functional equivalence with the original piece of software. The level of confidence in an exact recapture of earlier representations decreases with the number of levels of abstraction reconstructed; nevertheless, the acceptability of a respecified design can be checked in this way.

2.2 Experiences from Reverse Engineering Projects

Complex Reverse Engineering projects for major software systems have been undertaken and successfully completed, applying the method described with computer assistance by the EPOS System /Laub87/, /LaLe86/. One typical example showing the feasibility of the approach was the TORNADO reverse engineering project, performed as a multi-company effort under the direction of West German MOD.

This project was undertaken to recapture requirements and design specifications for the main avionics computer of this fighter aircraft, and for software elements involving its control, executive, navigation and targeting systems.

The problems confronting engineers were typical of the difficulties encountered in any Reverse Engineering project. During the original development, the avionics package consisted of software requirements specifications, design specifications, and programming code. This documentation was originally developed for the most part manually and without any form of clear, consistent interactive relationship between the elements. During the extended life of the project, programming code was modified on innumerable occasions to meet maintenance or partial upgrade requirements. This modified code was accompanied by varying degrees of annotation and documentation, and often was without complete reference to software requirements or design specifications. As time went on, the design specification documents lagged well behind actual code implementation. The reverse engineering project was undertaken to realign software requirements documents, design specifications and code so as to obtain a firm basis for further maintenance and migration to other projects.

Existing assembler program code, along with limited annotations and program commands, as available, was used for Reverse Engineering. The code was then used to tie into old design specifications. In this fashion the specification tree as well as the requirements documents were recreated, so that at the end of the project a complete design specification was accomplished. The finished specifications and the inherent algorithms and other avionics-specific concepts are now available for maintenance and reuse in next-generation avionics projects.

A recently completed project for NASA[1] dealing with the possibilities of automated Reverse Engineering consisted of three main subtasks:

- development of a systematic Reverse Engineering Methodology,
- (manual) application of this method to a selected piece of existing software, and
- identification of a concept of additional tool support for (further) automation of the Reverse Engineering process.

The project was successfully completed, and led to a systematic methodology with a clear identification of the necessary steps as outlined in the previous chapter. The methodology also identifies the underlying assumptions and prerequisites for each step, as well as clearly defined end results. The Reverse Engineering example consisted of a 20,000 lines-of-code FORTRAN program from NASA's Deep Space Network program, a package with "near real-time" requirements. The Reverse Engineering of the specification identified errors, inconsistencies and missing items in the reference documentation, and the regenerated FORTRAN source code was verified to be functionally equivalent to the original code.

[1] NASA-980, "Reverse Engineering for Information Systems," concluded 30 July 1987.

3. (Final) Transition to Ada

The reverse engineered design can be the starting point for a complete regeneration of the software in Ada, a process for which the technology exists and automatic tool support is readily available (e.g., /LeZe87/).

However, one of the advantages of the described transition method over direct code-to-code translators is that higher levels of the design are accessible; through structural improvements directed towards Ada as the target language, the powerful language constructs of Ada can now be utilized. Therefore, besides avoiding the restrictions of non-meaningful names (e.g., only 6 characters in FORTRAN) /Sant86/, restructuring at different levels of the design is now feasible. This may include, e.g.,

- (high-level) synchronization
- restructuring of low-level (unstructured) control flow
- using data record structures and other powerful techniques to group related data objects. (Figure 4 shows an example of a redefined record structure which consisted of unrelated variables/array in FORTRAN).

Code generation can then produce all Ada packages from the design automatically, with correct program structure, data and type definitions, exception handling, tasking and statements for sequential and parallel control. This usually represents about 75-90% of the Ada code. At the lowest design level, mappings of I/O formats and/or specific library calls usually require additional but straightforward transformations. Illustrated in Fig. 5 is an I/O statement in FORTRAN, the low-level design documentation, and its resulting conversion to Ada.

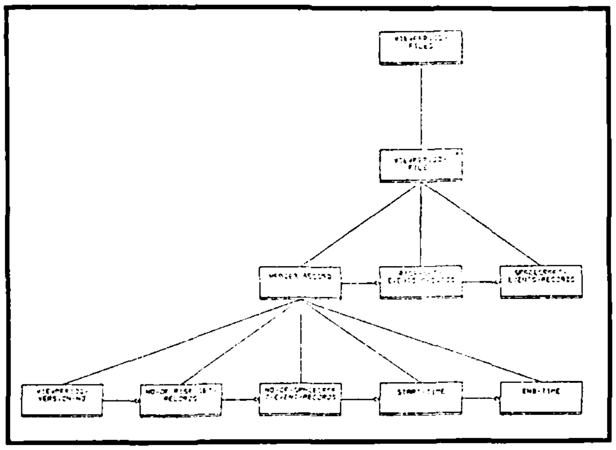
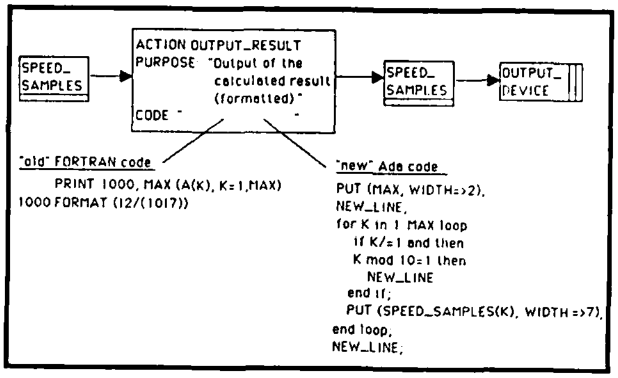


Fig. 4: Documentation of a restructured record structure

4. Computer-aided Support Environment

To perform the described steps in the transition effectively, computer-aided support is necessary. By first summarizing the characteristics which a development support environment ought to have for top-down Ada development, we can highlight the additional specific concepts and features necessary to perform a transformation using the Reverse Engineering approach.



```
      PRINT 1000, MAX (A(K), K=1,MAX)
1000 FORMAT (I2/(10I7))
```

```
PUT (MAX, WIDTH=>2),
NEW_LINE,
for K in 1 MAX loop
   if K/=1 and then
   K mod 10=1 then
      NEW_LINE
  end if;
  PUT (SPEED_SAMPLES(K), WIDTH =>7),
end loop;
NEW_LINE;
```

Fig. 5: Example for transformation of a low-level design/code block (see also /Sant86/).

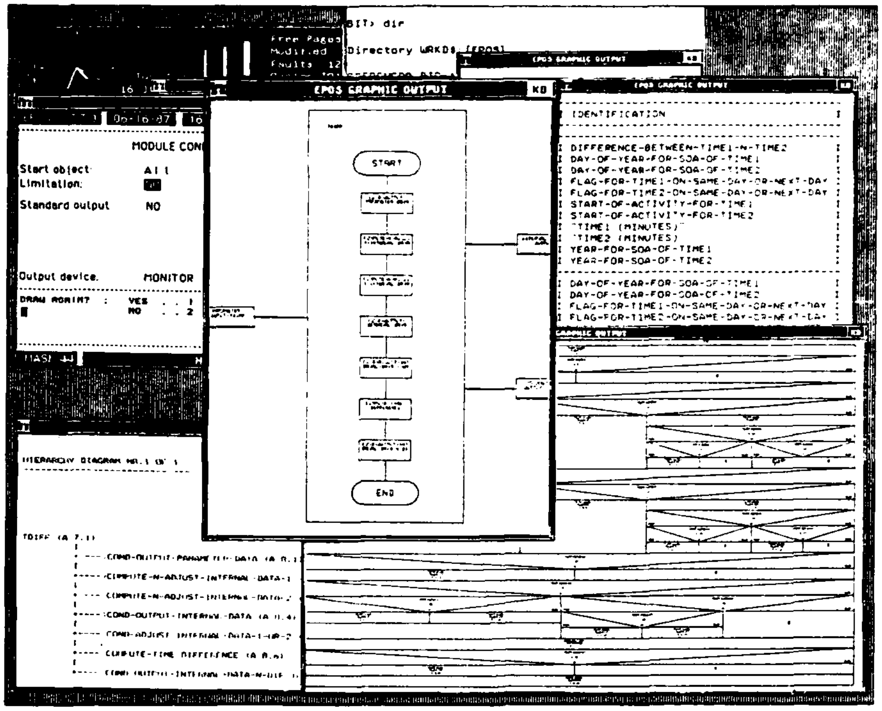
In a development support environment used for first designing the software in a (mostly) target-language-independent form, then using a code generator for reaching the source code level, the key elements are

- Support for various levels of abstractions
- Availability of a "rich" design language /Lemp87/.

Both characteristics are necessary to ensure that software designs of earlier developments can be regenerated in Ada, including the option of performing the restructuring possibilities (described in the previous chapter) within the same support environment. Beyond this, there are very specific requirements for a computer-aided Reverse Engineering environment, mainly in the following areas:

- Source code transformation tools
  To automatically extract (low level) design specifications from the source code and generate a design model in the design database.
- Comprehensive documentation and visualization features from various points of view
  To determine the functionality of a piece of software recaptured from mainly source code, all aspects (tasking, i/o, control flow, data flow, etc.) must be presentable in a correlated manner. Figure 6 shows an example from an actual project visualizing input/output and controlflow to assist in deducing an overall abstraction.
- Databased query functions
  To allow either general search or confirmation of a specific functionality hypothesis, easy-to-use query methods of the specification must be available.
- Code generator facilities for different programming languages
  In addition to a powerful Ada source code generator, mechanisms to regenerate code in the original programming language(s) for verification purposes are necessary.

Additional requirements apply for (re-)establishing high-level requirements specifications and their interrelations with the design. To assist in the highly interactive functionality determination of all pieces of software, knowledge-based support can advise the reverse engineering personnel of expected or suggested functionality (based on a knowledge of application-typical functions and their characteristics). This knowledge-based support can even help in automating portions of this reverse engineering activity.[2] Figure 7 summarizes the necessary tool components.[2]



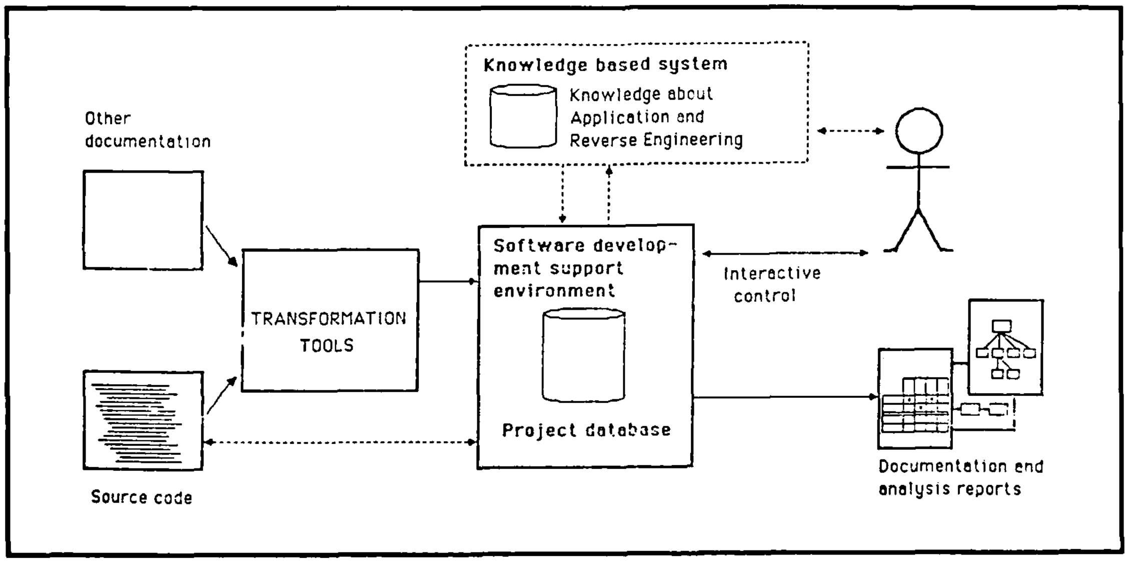
```
BIT> dir

Directory WRKD$:[EPOS]
```

```
Start object:        All
Limitation:
Standard output      NO

Output device.       MONITOR
DRAW MORINT? :       YES . . 1
                     NO  . . 2
```

```
I IDENTIFICATION                                  I
I DIFFERENCE-BETWEEN-TIME1-N-TIME2                I
I DAY-OF-YEAR-FOR-SOA-OF-TIME1                    I
I DAY-OF-YEAR-FOR-SOA-OF-TIME2                    I
I FLAG-FOR-TIME1-ON-SAME-DAY-OR-NEXT-DAY          I
I FLAG-FOR-TIME2-ON-SAME-DAY-OR-NEXT-DAY          I
I START-OF-ACTIVITY-FOR-TIME1                     I
I START-OF-ACTIVITY-FOR-TIME2                     I
I "TIME1 (MINUTES)"                               I
I "TIME2 (MINUTES)                                I
I YEAR-FOR-SOA-OF-TIME1                           I
I YEAR-FOR-SOA-OF-TIME2                           I
I DAY-OF-YEAR-FOR-SOA-OF-TIME1                    I
I DAY-OF-YEAR-FOR-SOA-OF-TIME2                    I
I FLAG-FOR-TIME1-ON-SAME-DAY-OR-NEXT-DAY          I
I FLAG-FOR-TIME2-ON-SAME-DAY-OR-NEXT-DAY          I
```

```
HIERARCHY DIAGRAM NO.1 OF 1

TDIFF (A 7.1)
    |
    +---- COND-OUTPUT-PARAMETER-DATA (A 8.1
    |
    +---- COMPUTE-N-ADJUST-INTERNAL-DATA-1
    |
    +---- COMPUTE-N-ADJUST-INTERNAL-DATA-2
    |
    +---- COND-OUTPUT-INTERNAL-DATA (A 8.4)
    |
    +---- COND-ADJUST INTERNAL-DATA-1-OR-2
    |
    +---- COMPUTE-TIME DIFFERENCE (A 8.6)
    |
    +---- COND-OUTPUT-INTERNAL-DATA-N-DIF
```

Fig. 6: Simultaneous visualizations of low-level design aspects to grasp functionality



Knowledge based system
Knowledge about Application and Reverse Engineering
Other documentation
TRANSFORMATION TOOLS
Software development support environment
Project database
Interactive control
Source code
Documentation and analysis reports

Fig. 7: Concept of computer-aided support environment for Reverse Engineering and regeneration of software in Ada

[2] In the NASA project mentioned above, the software engineering environment EPOS was used as the computer support kernel in conjunction with manual transformations; tools for automating formal code analysis and knowledge-based support are in development.

Conclusion

The transition from existing software written in conventional programming languages to new projects now begun using the Ada technology often requires more than a code-to-code translator. For parts of the software to be successfully reused, to be maintainable and an integral part of the new software product, the reused portions must meet the same quality and documentation standards as does the newly developed software.

Therefore the transition from existing code in FORTRAN, CMS-2, HAL/S, etc. should be made using the Reverse Engineering process of recapturing higher-level design specifications, and subsequently regenerating Ada source code. This provides the possibility of restructuring the software at the design level to utilize powerful constructs of Ada, and it enables automatic generation of complete and consistent documentation from the design specification. The approach has been shown successful in projects using the computer-aided support of the software engineering environment EPOS. Although not fully automatable, the method provides a cost-effective and practical solution for further utilization of existing software which incorporates years of application experience in concepts, algorithms and heuristics.

Acknowledgement

The author owes special thanks to the project team performing the NASA Reverse Engineering project, Mr. Ernesto Gomez in particular. The review of the paper was done by Dr. Alex Rainer; her valuable suggestions contributed to the overall cohesiveness.

References:

/ABCC87/ Antonini, P.; Benedusi, P., Cantone, G.; Cimitile, A.: Maintenance and Reverse Engineering: Low-level design documents production and improvement. Proc. IEEE Conf. on Software Maintenance, Austin, Sept. 1987, IEEE Computer Society Press, pp. 91-100.

/Aran86/ Arango, G.: "TMM: Software Maintenance by Transformation." *IEEE Software*, Vol. 3, No. 3 (May 1986), pp. 27-39.

/BABK87/ Burton, B.; Aragon, R.; Bailey, S.; Koehler, K.; and Mayes, L.: "The Reusable Software Library". *IEEE Software*, July 1987, pp. 25-33.

/BrCr86/ Britcher, R.N.; Craig, J.J.: "Using Modern Design Practices to Upgrade Aging Software Systems." *IEEE Software*, Vol. 3, No. 3 (May 1986), pp. 16-24.

/HKPT87/ Huijsman, R.D.; Kavan Katwijk, J.; Pronk, C.; Toetenel, W.: "Translating Algol 60 Programs into Ada™." *Ada letters*, Vol. VII, No. 5 (Sept/-Oct. 1987), pp. 42-50.

/LaLe86/ Lauber, R.; Lempp, P.: EPOS Overview. New York: SPS Software Products & Services, Inc., 1986.

/Laub87/ Lauber, R.: "Automated Software Production." AIAA/NASA International Symposium on Space Systems in the Space Station Era, Washington D.C., June 22-23, 1987.

/Lemp87/ Lempp, P.: "An Environment to Promote Software Reusability at the Design Specification Level." Proc. 5th Pacific Northwest Software Quality Conf., Portland, OR, Oct. 1987.

/LeZe87/ Lempp, P., Zeh, A.: "Developing Ada™ Software using the Software/Hardware Production Environment EPOS." SDIO Ada Conference and Tools Fair, Washington D.C., Jan. 1987.

/McNi86/ McNicholl, D.: "Common Ada™ Missile Packages." Proc. Nat. Conf. on Software Reusability and Maintainability, Tysons Corner, Virginia, Sept. 1986.

/Sant86/ Santhanam, V.: "A Practical Approach for Translating FORTRAN Programs to Ada." Proc. 4th Annual Nat. Conf. on Ada Technology, 1986, pp. 142-148.

The Author:

Peter Lempp is a group leader of Software Engineering and Manager of International Projects. His major areas of interest are software engineering, integrated Ada support environments, project management of software projects, and software quality assurance.

Mr. Lempp received an advanced degree in Electrical Engineering from the University of Stuttgart, Germany, in 1983. He did graduate work at Northwestern University, Evanston, Illinois in 1981, and has recently submitted his Ph.D. thesis in Electrical Engineering/Computer Science at the University of Stuttgart. Mr. Lempp has done significant R & D work in the area of software engineering environments, including work as project leader for the EPOS Support Environment, a project team leader in a NASA Reverse Engineering Project, and in the European ESPRIT program in 1984-85.

Mr. Lempp is a member of the IEEE, the German Computer Science Society (GI) and the ACM (SIGSoft, SIGAda).

Mailing Address:
SPS Software Products & Services, Inc.
14 East 38th Street, 14th Floor
New York, NY 10016

Telephone: (212) 686-3790

Paper 10.4, "Translating Embedded Software to Ada: Practice and Experience" (pages 339–354), was withdrawn

SOFTWARE DEVELOPMENT STANDARDS: NECESSITIES, LIMITATIONS, AND OPPORTUNITIES

George W. Macpherson

SofTech, Inc.

Abstract - The size and complexity of the Department of Defense mission require that, to the greatest extent practical, resources be standardized for effective and efficient management and operation. DoD software development, especially with the advent of Ada, is a prime example of the application of standardization. This paper reviews the structure of the MIL and DoD standards that apply to software development with special attention given to DoD-STD-2167 and its relationship with the Ada MIL/ANSI-1815A. Finally, in spite of the complexities of Ada, this paper suggests a method which could reduce costs of initial software delivery by proper use of the language.

The Role of Standards

Over the centuries of scientific evolution and development, we have learned that in most areas, a degree of standardization is very beneficial, if not absolutely essential. Mutual acceptance of standards allows us to communicate our ideas clearly with no ambiguity, and to build separate parts of a large system so these parts "fit together", physically, electronically, and functionally. Some standards are quite absolute: the standard meter, the standard coulomb, the standard second, and so on. Other standards are less absolute and provide certain options; examples of these are ADCCP and TCP/IP protocols. And of course, some standards are quite subjective, we have all seen standards that call for "good structure", "logical organization", and so on.

The software standards we work under cover the full spectrum of this specificity. The developers of software standards carry a heavy burden in finding the correct balance between absolute requirements and general guidance, which will provide the optimal software development environment. Standards which are specific and iron-clad cannot anticipate break-throughs in our rapidly advancing technology and tend to stifle innovation. If standards are too general and subjective, the goals and benefits of standardization can be completely lost.

Two Significant Events

Recently two extremely significant events in the area of software standards have occurred. The first was the 22 Jan 1983 publication of ANSI/MIL-STD 1815A, the Ada Programming Language. The second was the 4 Jun 1985 publication of the Software Development Standards (SDS) Documentation Set (which includes DoD-STD-2167). The adoption of the Ada language reflects judgement that while individual and separate languages may have unique advantages for specific applications, the chaos in portability and maintainability created by this multiplicity of languages far outweighs any individual advantage. Although no single language might be best for every application, it was decided that a standard language incorporating the best of software technology, but including many compromises and possibly a few imperfections, would serve far better, particularly in the area of mission-critical embedded systems. The adoption of the SDS Documentation Set marked a milestone of cooperation among the military services. Now all the services deal with software under a single set of standards. To the great relief of both contractors and reviewing officials, volumes of overlapping and potentially contradictory standards and directives have been streamlined into a far more workable and efficient system. But while the SDS Documentation Set cleared up many problems, the Ada language introduced totally new programming concepts, opportunities for more efficient software development, and substantial reduction of life cycle costs. If these software development advantages of Ada are to be properly exploited, we must be both perceptive and innovative in our applications of standards.

Structure of the SDS Documentation Set

Those readers who have spent long years developing software under the military standards system may wish to skip this section, but for those who are relatively new to the process, a quick overview may be helpful. The SDS Documentation Set contains the following:

1. Joint Regulation (un-numbered), "Management of Computer Resources in Defense Systems". This regulation establishes policy for the acquisition, management, and support of Mission-Critical Computer Systems (MCCS) software during all phases of the system life cycle. Perhaps the most important facet of the regulation is that it establishes a software authority and responsibility for all the armed forces in a single entity.

2. DoD-STD-2167, "Defense System Software Development". This standard establishes uniform requirements for software development that are applicable throughout the system life cycle. The standard has been under nearly constant review and revision since its first publication, and in this author's view, one of the important and most over-looked aspects of this document is the spirit of the standard which is expressed in paragraph 2 of the Forward of the 4 September 1987 draft, which reads, "This standard is not intended to specify or discourage the use of any particular software development method. The contractor is responsible for selecting software development methods that best support the achievement of contract requirements". It should be noted that Appendix E of 2167 is guidance for tailoring the standard to support the unique characteristics of any individual software development.

3. MIL-STD-483, "Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs". This standard with its very broad scope, is being phased out of the software picture. In particular, its Appendix VI, "Computer Software Configuration Item Specification", has been replaced by two Data Item Descriptions (DIDs).

4. MIL-STD-490, "Specification Practices". This is where the terms "B5 Specification" and "C5 Specification" came from. Like MIL-STD-483, it is being phased out of software development and both B5 and C5 specifications have taken the form of sets of Data Item Descriptions.

5. MIL-STD-1521, "Technical Reviews and Audits for Systems, Equipments and Computer Programs". With 1521, the standard system becomes more specific. This standard describes ten reviews and audits that may be conducted during the course of software (or hardware) development. In software, probably the most important of these are the Preliminary Design Review (PDR) and Critical Design Review (CDR). But 1521 does far more than prescribe an agenda for a review meeting. This standard, along with the Data Item Descriptions, dictate what kind of design information is to be presented. PDR and CDR must be successfully completed if development is to continue, and typically 40 percent or more of development resources are expended by the time CDR is completed. Software design and preparation for PDR and CDR are strongly influenced by 1521 and the DIDs. Application of the standard and the DIDs becomes a critical factor in the software development process.

6. Data Item Descriptions. By far the greatest direct influence of the military standards system on software development is through the Data Item Descriptions. The guidance given by the previously described standards is much more general and in some cases subjective. The DIDs give a specific format for expressing design, and the method for expressing a design has a surprisingly large influence on how the design is accomplished. There are 24 DIDs in SDS Documentation Set, ranging from the Operational Concept Document to the Detailed Design Document. The thrust of the paper focuses around production of the Software Top-Level Design Document, DI-MCCR-80012.

Software Development Philosophy Changes Direction. Before proceeding into a proposed approach to preparation of a Software Top-Level Design Document, it might be well to step back and get perspective by reviewing how software development has emerged from the very start. Back in ENIAC days, when computers were designed to automate the operations of desk calculators, programs were written in binary with something like "101" standing for Clear and Add, and perhaps "110" for Store. Memory storage locations were referenced by their absolute binary address and an entire computer program consisted of nothing more than lots of ones and zeros. The arrival of assembler languages relieved the programmer of keeping track of numerical addresses and allowed use of mnemonics for machine instructions. Then a program consisted of statements like CLA A and STO B; the code was far easier to read and much quicker to write. Compiler languages further abstracted the programming process so that the programmer was no longer required to understand the details of machine instructions, and programs consisted of mathematical formula-like statements such as X=(-B+SQRT(B**2-4*A*C))/(2*A). In all these developments, the thrust was to make it easier and quicker for a programmer to get his program operating. This worked quite well as long as software systems were fairly small and only a few programmers were involved in writing systems, which ran on a single computer.

As software systems grew in size, involving large numbers of programmers and multiplicities of computers, problems began to emerge which are now generally recognized as the "Software Crisis". Software reliability became very questionable. When it became necessary to modify software to meet new requirements, or to move it to a new and more powerful computer, we often found after expending a great deal of time and resources that it was cheaper to start again from scratch rather than to understand, untangle and modify the existing software. Software costs skyrocketed, deliveries were late, and all too often after spending a great deal of money, the software never became operational. We discovered that the language philosophy of simply making programs easier and quicker to write was getting us into serious trouble and things had to change. We discovered that we were expending far more effort in repair and modification of software than in design and production. We discovered that while software is written once, it is read and analyzed perhaps hundreds of times. Software languages and philosophy had to change to address the software task from the full life cycle point of view.

Computer hardware development continued to advance at an astounding rate while software development technology crept forward with painful slowness. To emulate the achievements of the hardware engineers, we created the term "Software Engineering". But creating a term does not necessarily create an engineering discipline. Dr. John Manley, former director of the Software Engineering Institute, is quoted as saying, "... although progress is being made, software engineering is still an aspiration...". The established branches of engineering have splendid mathematical models to work with such as Newtonian Mechanics, Maxwell's Equations, and the periodic table of the elements. In software, we're still groping. Dr. Frederick Brooks, author of the classic book "The Mythical Man-Month"[1], gives insight on the problem in an article entitled "No Silver Bullet"[2] in the April 1987 issue of Computer Magazine. Dr. Brooks suggests that we are unlikely to find a silver bullet to cleanly and quickly slay the software crisis werewolf, and that a more tedious process will probably be required. In noting the discrepancy between the advancement rates of hardware and software development technologies, he suggests that perhaps the two problems are different, and software development technology might be more like another field, medicine, which had to take a dramatic change in direction. There came a time when that profession transitioned from a demon theory to a germ theory. That transition did nothing to improve surgical productivity, the business of washing hands and sterilizing instruments was time consuming and costly, but the overall effect on the very literal life cycle was gratifying.

In our change of direction in software development philosophy,we are endeavoring to employ engineering principles in software development. We get encouragement from another statement by Dr. Manley, "Ada is a vehicle for introducing software engineering methods into use". It was mentioned earlier that the method of expressing a software design is closely related to the method with which the design is produced. The next section will propose a method for expressing a design (thereby impacting on design production) which is a departure from traditional methods.

Opportunities

The opportunity to reduce software development costs hinges on recognizing, and taking advantage of the fact that Ada is itself a design language. In a nutshell the opportunity is this: By expressing some of the design information required by the DIDs in Ada, rather than charts and tables, software development costs can be reduced. The Appendix in this paper gives an example of how this method could be applied to portions DI-MCCR-80012, the Software Top-Level Design Document (STLDD). The STLDD requires that certain design information concerning the software, such as range of values, data type, precision, etc., be presented. The STLDD goes on to give examples of how this design information might be presented in the form of tables (Figures 1,2). The appendix shows how all the design information presented in these example tables can be expressed in Ada, (Figure 3) and we will examine reasons why this simple change in design information format can reduce software development costs. The assertion here is not just that software designated to be implemented in Ada could benefit from this method, but also that using this method in an Ada environment can reduce development cost compared with producing the same software in another language such as FORTRAN. This assertion should be met with skepticism. "Common Sense" would tell us that since Ada is more complex and more demanding on the programmer, it is only natural to expect that more resources would be required for initial delivery and we would hope that this additional effort would be more than compensated for by the benefits of reliability, modifiability, and portability. At this point, it is well to recall two statements by Grady Booch in his well known book, "Software Engineering with Ada"[3] First, "...the phases of design/code/test are no longer distinct; rather, they should form an iterative process as each level of the solution..." and second, "Etch the following five words in your mind: Ada is a design language."

In reflecting on the first statement, it has been our general practice under the military standards system to write a complete and detailed design before any coding is done. This is possibly an over-reaction to the days when coding was vigorously pursued before any sound designing had been done. But this practice is not consistent with standard engineering practices. In the other engineering disciplines, promising design ideas are implemented in models, breadboards, mock-ups, and prototypes. Those models are rigorously tested, and the results at these tests confirm or reject the feasibility of the design concept, and give direction on how further design should proceed. All too often in software, the infeasibility of a design is discovered only at the late and expensive system testing phase.

Concerning Mr. Booch's second remark, it took this author some time to appreciate just what "Ada is a design language" meant. That Ada is a design language does not imply that it is by itself the only tool required to produce a software design. Structured Analysis and Design Techniques (SADT), Object Oriented Design (OOD) and other tools are extremely useful, particularly at the initial phases of design. But Ada is also a design tool, and is particularly useful in designing interfaces and data dictionaries. Another factor concerning Ada as a design language is that there is a lingering misconception in the software community that a program design language (PDL) and an implementation language are necessarily separate and distinct things. Before Ada arrived this was true. Now it is not true. There is another important issue concerning use of Ada as a design language. We are all aware that it is possible to get bogged down in minute details during design and lose sight of the complete design picture. On the other hand, there is an insidious temptation, when faced with a controversial, important, and difficult design decision, to be lulled into designating that decision as an "implementation detail". In these cases the designer might be very well served by coding, compiling, and testing skeletal versions of the alternatives.

Returning to the proposed method, the Appendix gives a detailed comparison of presenting design information in a compiled Ada format versus a table format, and shows specific examples of how the Ada format can avoid problems which could develop in the table format. The ways in which the Ada format can reduce software development costs may be summarized as follows:

1. The Ada design information is validated for consistency, language legality, and dependencies by the Ada compiler. The design information in table format can be checked only by manual methods, which may miss design flaws and result in very expensive repairs in the test phase.

2. As software development proceeds, the validated design information becomes part of the operational software, with no risk of error in translating design information in the form of charts, tables, and PDL into implementation code. Thus, coding and testing costs are reduced.

3. The task of Quality Assurance in the areas of range of values, constraints, and data protection, is greatly simplified, and thereby less costly.

4. A great deal of the design information is embedded in the software. As modifications, such as changes in parameters or range of values are made in the developing software, these design changes are automatically incorporated in the Ada specification sections, and the software acquires a substantial degree of self-documentation. Thus documentation costs are reduced.

Let me reiterate that what is being proposed here is not an entire software development methodology. It is a technique which may be embedded in any of a number of methodologies to enhance development efficiency. The example in the Appendix gives an alternative method for expressing some of the design information required by the Data Item Description, DI-MCCR-80012, Software Top-Level Design Document, dated 30 June 1986. This DID is currently under revision and the present draft no longer deals with Top-Level Computer Software Components, TLCSCs (CSCs only) and the new draft does not give a sample CSC output table. The changes however, do not effect the concept of the proposed method. Also, this method is equally applicable to DI-MCCR-80027, the Interface Design, Document and DI-MCCR-80031, the Software Detailed Design Document.

Conclusion

Standards perform a necessary and useful function in every scientific endeavor. This paper has presented an Ada application of standards which hopefully could improve software development efficiency. Standards are not, nor are they intended to be, a substitute for the creative process. There is always a danger that interpretations of standards will restrict or confine innovative methods. In making decisions about use of new methodologies, there is often a deadly fear that the new methods may be "non-compliant", and a narrow, "safe" interpretation of the standards is chosen. But here is something we all know very well and must always remember: Full compliance with the regulations and standards is no excuse for failure.

APPENDIX

AN ALTERNATE METHOD FOR DEVELOPING THE SOFTWARE TOP-LEVEL DESIGN DOCUMENT, DI-MCCR-80012.

This section suggests an alternate method of expressing the design information expressed tn Tables IV,V,VI, and VII of DI-MCCR-80012. These tables are entitled:

Table IV - Sample global data definition table

Table V - Sample TLCSC X input table

Table VI - Sample TLCSC X local data definition table

Table VII - Sample TLCSC X output table

This alternate method is to express that information by means of a package called EXAMPLE_TLCSC_GLOBAL_DATA and a procedure called EXAMPLE_TLCSC_X. A line-by line explanation of these two compilation units follows: In all cases, the columns entitled "IDENTIFIER" in Tables IV through VII have been ignored, since the columns entitled "DESCRIPTION" contain information which seems more appropriate to use as Ada identifiers. These examples are not offered as "good" Ada, but only as an example of alternate methodology.

EXAMPLE_TLCSC_GLOBAL_DATA.

This package contains not only the design information from Table IV, but also the design information from Tables V and VII which is declared to be GLOBAL.

Line 2 - MILES is given as a derived type only to demonstrate that this is a design option.

Line 3 - Table IV does not give a range of values for COORDINATES, but a good Ada design should invoke a range constraint.

Line 4,5 - It would be quite simple to express PRIORITY_LEVELS as a range of integer values in Ada, but this would be contrary to the fundamental Ada philosophy of strong typing. A user defined enumeration type is clearly called for.

Lines 6-10 and 18 - This section clearly demonstrates a great advantage in expressing design information in compiled Ada code. The design information for "ALIST" in Table IV is very fuzzy and open to many different interpretations. Lines 6-10 and 18 give one of these interpretations in precise and unequivocal form. The advantage here is that at design review, this precise record and array definition may be examined, and revised if necessary to meet system requirements. If the design of "ALIST" is given in Table IV were allowed to stand until test time, some very unpleasant surprises and expensive repairs might result.

Line 12 - SPEED is an input parameter of Table V and an output parameter of Table VII Therefore its type must be visible to both TLCSC_X and the calling software, and the type definition is placed in the GLOBALS package. Note also that Tables V and VII give SPEED a range of 1.0..10_000.0 and the GLOBALS package gives a range of only 1.0..1_000.0. This is because the compiler used in preparing this example is simply not capable of producing a fixed point type with a delta of 0.000_005 and a range of 1.0..10_000.0. This is another example of detecting a design problems at an early stage, when it is far less expensive to resolve.

Lines 13,14 & 24 - Table VII lists a global output array with the identifier "COORDS" just as does Table IV. But the precision of the two "COORDS" is different. Therefore, in this example, the identifier "B_MILES" and "B_COORDINARE_ARRAYS" are used. This is another example of early detection of design problems.

Lines 21-23 - These are the global input variables from Table V.

EXAMPLE_TLCSC_X.

This procedure skeleton defines the local data of Table VI, and the input and output parameters of Tables V and VII. Of course the global inputs and outputs are defined in the GLOBALS package.

Line 1. Visibility to the GLOBALS package is required for the type of the procedure parameter.

Lines 2,3 - Tables V & VII indicate that SPEED is an in out parameter.

Line 10 - Same "range" problem detection as line 12 of the GLOBALS package.

Line 12 - PIXEL_STATUS is made a user defined enumeration type to make it absolutely clear to any reader exactly what is meant by PIXEL_STATUS values.

Line 17 - The identifier here has been changed slightly from Table VI to make its purpose more clear. Note that the use of the Ada "constant" feature simplifies the QA task of assuring proper conversion.

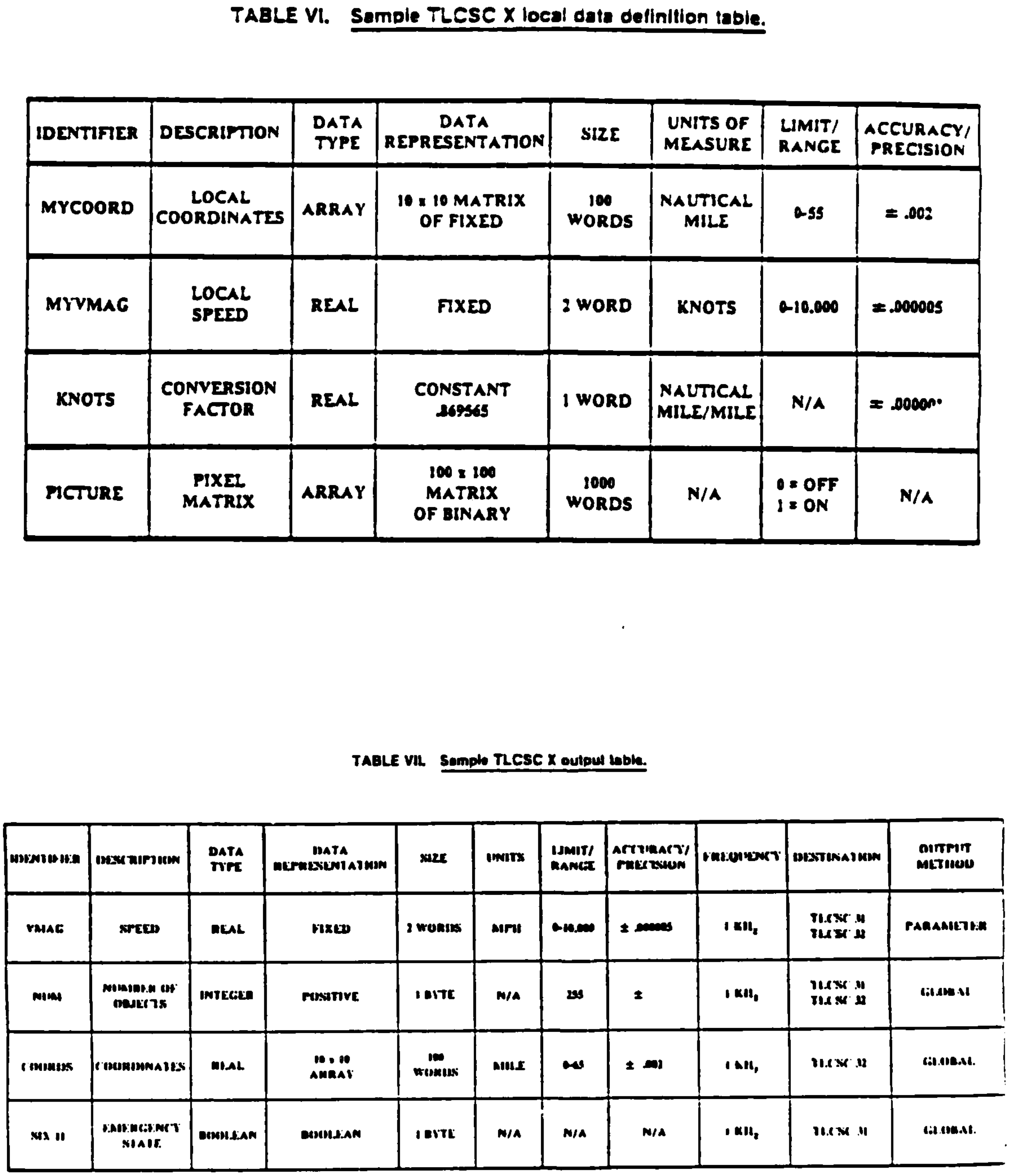


TABLE VI. Sample TLCSC X local data definition table.

IDENTIFIER	DESCRIPTION	DATA TYPE	DATA REPRESENTATION	SIZE	UNITS OF MEASURE	LIMIT/ RANGE	ACCURACY/ PRECISION
MYCOORD	LOCAL COORDINATES	ARRAY	10 x 10 MATRIX OF FIXED	100 WORDS	NAUTICAL MILE	0-55	± .002
MYVMAG	LOCAL SPEED	REAL	FIXED	2 WORD	KNOTS	0-10.000	±.000005
KNOTS	CONVERSION FACTOR	REAL	CONSTANT .869565	1 WORD	NAUTICAL MILE/MILE	N/A	± .00005
PICTURE	PIXEL MATRIX	ARRAY	100 x 100 MATRIX OF BINARY	1000 WORDS	N/A	0 = OFF 1 = ON	N/A

TABLE VII. Sample TLCSC X output table.

IDENTIFIER	DESCRIPTION	DATA TYPE	DATA REPRESENTATION	SIZE	UNITS	LIMIT/ RANGE	ACCURACY/ PRECISION	FREQUENCY	DESTINATION	OUTPUT METHOD
VMAG	SPEED	REAL	FIXED	2 WORDS	MPH	0-10,000	± .000005	1 KHz	TLCSC 31 TLCSC 32	PARAMETER
NUM	NUMBER OF OBJECTS	INTEGER	POSITIVE	1 BYTE	N/A	255	±	1 KHz	TLCSC 31 TLCSC 32	GLOBAL
COORDS	COORDINATES	REAL	10 x 10 ARRAY	100 WORDS	MILE	0-65	± .001	1 KHz	TLCSC 32	GLOBAL
SIX II	EMERGENCY STATE	BOOLEAN	BOOLEAN	1 BYTE	N/A	N/A	N/A	1 KHz	TLCSC 31	GLOBAL

Figure 1 - Tables VI and VII from DI-MCCR-80012

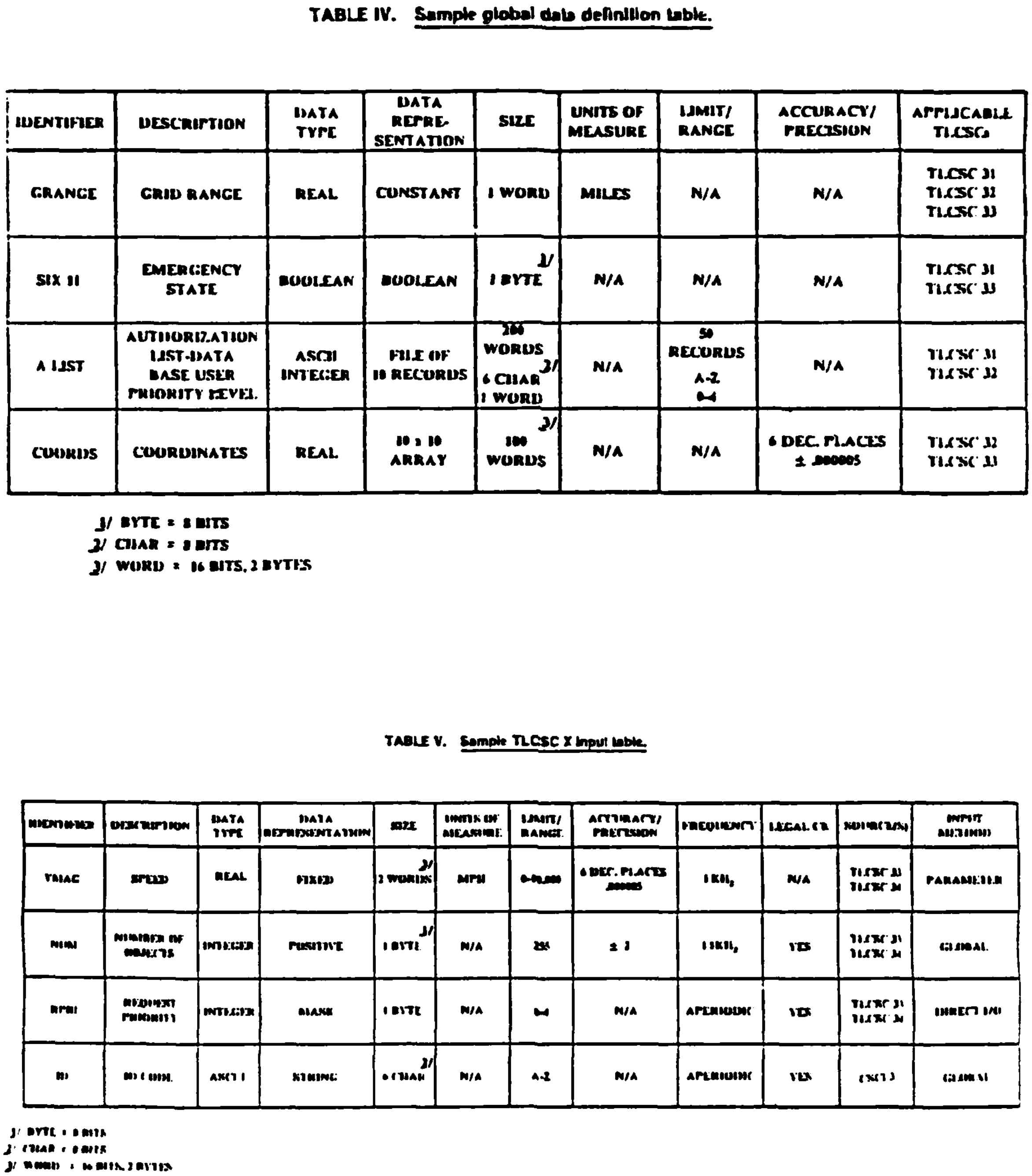


TABLE IV. Sample global data definition table.

IDENTIFIER	DESCRIPTION	DATA TYPE	DATA REPRE-SENTATION	SIZE	UNITS OF MEASURE	LIMIT/ RANGE	ACCURACY/ PRECISION	APPLICABLE TLCSCs
GRANGE	GRID RANGE	REAL	CONSTANT	1 WORD	MILES	N/A	N/A	TLCSC 31 TLCSC 32 TLCSC 33
SIX 11	EMERGENCY STATE	BOOLEAN	BOOLEAN	1/ 1 BYTE	N/A	N/A	N/A	TLCSC 31 TLCSC 33
A LIST	AUTHORIZATION LIST-DATA BASE USER PRIORITY LEVEL	ASCII INTEGER	FILE OF 10 RECORDS	200 WORDS 2/ 6 CHAR 1 WORD	N/A	50 RECORDS A-Z 0-9	N/A	TLCSC 31 TLCSC 32
COORDS	COORDINATES	REAL	10 x 10 ARRAY	3/ 100 WORDS	N/A	N/A	6 DEC. PLACES ± .000005	TLCSC 32 TLCSC 33

1/ BYTE = 8 BITS
2/ CHAR = 8 BITS
3/ WORD = 16 BITS, 2 BYTES

TABLE V. Sample TLCSC X input table.

IDENTIFIER	DESCRIPTION	DATA TYPE	DATA REPRESENTATION	SIZE	UNITS OF MEASURE	LIMIT/ RANGE	ACCURACY/ PRECISION	FREQUENCY	LEGAL CK	SOURCE(S)	INPUT METHOD
VMAC	SPEED	REAL	FIXED	3/ 2 WORDS	MPH	0-99,000	6 DEC. PLACES .000005	1 KHz	N/A	TLCSC 33 TLCSC 34	PARAMETER
NUM	NUMBER OF OBJECTS	INTEGER	POSITIVE	1/ 1 BYTE	N/A	255	± 1	1 KHz	YES	TLCSC 31 TLCSC 34	GLOBAL
RPRI	REQUEST PRIORITY	INTEGER	RANK	1 BYTE	N/A	0-9	N/A	APERIODIC	YES	TLCSC 31 TLCSC 34	DIRECT I/O
ID	ID CODE	ASCII	STRING	2/ 6 CHAR	N/A	A-Z	N/A	APERIODIC	YES	CSCI 3	GLOBAL

1/ BYTE = 8 BITS
2/ CHAR = 8 BITS
3/ WORD = 16 BITS, 2 BYTES

Figure 2 - Tables IV and V from DI-MCCR-80012

```
package EXAMPLE_TLCSC_GLOBAL_DATA is
    type MILES is new float;
    type COORDINATES is delta 0.000_005 range 0.0..1.0;
    type PRIORITY_LEVELS is (ROUTINE,PRIORITY,IMMEDIATE,FLASH,
                             EMERGENCY);
    type AUTHORIZATION_LISTS is
        record
            USER           : string (1..6);
            PRIORITY_LEVEL : PRIORITY_LEVELS;
        end record;
    type COORDINATE_ARRAYS is array (1..10, 1..10) of COORDINATES;
    type SPEEDS is delta 0.000_005 range 0.0..1_000.0;
    type B_MILES is delta 0.001 range 0.0..65.0;
    type B_COORDINATES_ARRAYS is array (1..10, 1..10) of B_MILES;
    subtype NUMBERS_OF_OBJECTS is positive range 1..255;
    GRID_RANGE             : MILES := 1000.0;   -- TLCSC 31,32,33
    EMERGENCY_STATE        : boolean;           -- TLCSC 31,33,X
    AUTHORIZATION_LIST     : array (1..10) of AUTHORIZATION_LISTS;
                                                -- TLCSC 31,32
    COORDINATE             : COORDINATE_ARRAYS;
    NUMBER_OF_OBJECTS      : NUMBERS_OF_OBJECTS;-- TLCSC X
    ID_CODE                : string (1..6);     -- TLCSC X
    REQUEST_PRIORITY       : PRIORITY_LEVELS;   -- TLCSC X
    B_COORDINATES_ARRAY    : B_COORDINATES_ARRAYS;  -- TLCSC_X
end EXAMPLE_TLCSC_GLOBAL_DATA;
```

```
with EXAMPLE_TLCSC_GLOBAL_DATA;
procedure EXAMPLE_TLCSC_X
          (SPEED : in out EXAMPLE_TLCSC_GLOBAL_DATA.SPEEDS) is
--
-- Local Variables
--
    type LOCAL_COORDINATES is delta 0.001 range 0.0..55.0;
    type LOCAL_COORDINATES_ARRAYS is array (1..10,1..10) of
                                            LOCAL_COORDINATES;
    type LOCAL_SPEEDS is delta 0.000_005 range 0.0..1_000.0;
    type CONVERSION_FACTORS is delta 0.000_001 range 0.0..1.0;
    type PIXEL_STATUS is (OFF,ON);
    type PIXEL_ARRAYS is array (1..100,1..100) of PIXEL_STATUS;

    LOCAL_COORDINATE_ARRAY : LOCAL_COORDINATES_ARRAYS;
    LOCAL_SPEED            : LOCAL_SPEEDS;
    MPH_TO_KNOTS           : constant CONVERSION_FACTORS := 0.869565;
    PIXEL_MATRIX           : PIXEL_ARRAYS;
begin
    null;
end EXAMPLE_TLCSC_X;
```

Figure 3 - An Alternate Method for Expressing the Design Information of Tables IV,V,VI and VII of DI-MCCR-80012, "Software Top Level Design Document"

Acknowledgements

I thank Christine Ausnit and Sara Natale for insights and helpful suggestions, and Doris Brown for technical production of this article.

REFERENCES:

1. F.P.Brooks, The Mythical Man-Month, 1975, Addison-Wesley, Reading, Mass., New York.

2. COMPUTER, F.P. Brooks, Vol.20 No.8, April 1987.

3. G.Booch, Software Engineering with Ada, 1983, Benjamine/Cummings, Menlo Park, Calif.

George W. Macpherson received the B.S. degree in Electrical Engineering from the USAF Institute of Technology, Wright-Patterson AFB, Ohio, and the M.S. Degrees in Electrical Engineering and Mathematics from the University of Michigan, Ann Arbor. He served four years on the Computer Science faculty of the Air Force Academy, where he wrote a textbook on the ALGOL programming language. He has extensive experience in C^3 and scientific software, and has originated and taught courses in the Ada language. He has served as Vice-Chairman of the Association for Computing Machinery Special Interest Group on Ada Software Development Standards Ada Working Group (ACM SIGAda SDSWG) since the group's founding. He is presently a SofTech principle investigator for Ada Independent Research and Development, and is responsible for Ada training at the SofTech Colorado Springs facility. Readers may write to G.W. Macpherson at SofTech, Inc., Suite 305, 5475 Mark Dabling Blvd., Colorado Springs, CO 80918

KALMAN FILTERS IN GENERIC ADA* PACKAGES

SHERRIE A. BRANYON, HENRY B. KNOWLES, JR., GEORGE K. HESS, JR.

MARTIN MARIETTA CORPORATION, P.O. BOX 5837, ORLANDO, FL 32855

ABSTRACT

Kalman filters are used in signal processing applications where system state variables and observation variables contain noise. This paper describes generic Ada packages for four types of Kalman filters: 1) conventional Kalman filter, 2) square root covariance filter, 3) square root information filter, and 4) Chandrasekhar square root filter. These packages are maintained in a library of reusable Ada components. The Kalman filter packages are supported by packages and subprograms which provide interactive user inputs, target motion simulation, mathematics, matrix and vector operations, performance analysis, and graphic displays. Examples of the packages and subprograms developed are included in this paper. An extensive list of references to papers and books on these subjects is provided.

The application of a reusable Kalman filter package in an embedded computer requires evaluation of the package from two aspects: filter performance and embedded computer utilization. Preliminary results of such evaluations are given. Experience in this area supports the concepts of reusability of Ada components and improvements of life-cycle productivity.

I. INTRODUCTION

One of the requirements for the design of the Ada programming language [1] was that it provide facilities which support software reuse. As observed by Booch [2], the benefits of applying reusable software components are that they leverage the talents of good developers, thus improving the quality of the software; they also reduce the amount of software needed, which reduces the cost of software development and accelerates software production. The Ada features which provide the strongest support for development of reusable software components are the packaging mechanism, generic facilities, and the concept of program libraries. In Ada, packages are used to encapsulate related entities, which can include constants, objects, types, or program units. The structure of the package, i.e., a specification and a body which can be separately compiled, provides a means of controlling what is visible to, and usable by, the outside user.

As noted in [3], the package construct promotes the idea of creating an industry of reusable software components, and the generic unit enhances the possibility of such a service. The strong typing rules of Ada could be a deterrent to writing reusable Ada parts if not for the generic facilities provided by the language. Often, the logic of a section of code is independent of the type on which it must operate. The generic mechanism allows the user to create a template describing the logic and then instantiate (create an instance of) the generic unit for the particular type on which the logic must operate. Combining these two features produces the generic package, an eminently suitable mechanism for creating reusable software components [4] which can be placed in an Ada parts library and linked to produce a software program.

The purpose of this paper is to describe the design and testing of generic Ada packages which make several Kalman filters available in a library of reusable components. Kalman filters (KF) are used extensively in modeling physical systems. Many such systems can be modeled adequately by linear difference equations [5] which relate system state variables at discrete time instants. At each discrete time, a set of noisy inputs are applied to the system and a set of noisy measurements are produced.

The filtering problem is to produce optimal estimates [6] to [10] of the system state variables using all available measurements. Optimal estimates are usually based on minimizing the mean square error between the true values of the state variables and their estimated values. In 1960, Kalman published [11] a new approach to the linear filtering and prediction problems. In simple KF implementations, the previous state estimate vector and error covariance matrix are combined with a new measurement vector to produce the desired new state estimate. There are now many excellent technical papers [12] to [18] and books [19] to [20] which the reader should consult for information on the theory and application of Kalman filters.

Kalman filter implementations in reusable library packages have been selected from extensive published literature on KF and practical experience in KF applications. In the development of large Ada programs for land, sea, and air target tracking with laser, radar, and infrared sensors, it is considered necessary to simulate system operations. Therefore, the development of KF packages has been augmented with the development of other Ada packages and procedures so that system simulations can be easily assembled from the library of reusable Ada components.

It is intended that the KF Ada packages and supporting Ada library components will find their primary use in digital computers "embedded" in missiles, aircraft, and land vehicles [21]. Because of the wide range of such applications, it is expected that Kalman filter components will be selected from the parts library, customized if necessary, and then fine-tuned based upon the timing and memory requirements of the particular application. The design of the packages has been influenced primarily by the need for readability, modifiability, and maintainability. The conventional Kalman filter, as defined below, has been run on an embedded processor, and performance analysis for that system is being conducted.

Detailed technical information on KF theory and other KF applications can be found in the references cited in this paper. The emphasis in this paper is on Ada features as they relate to Kalman filter programs.

*Ada is a registered trademark of the United States Government, Department of Defense, Ada Joint Program Office (AJPO)

This paper is divided into six sections. Section II contains issues for reusable software components and the effects of those issues on the Kalman filter design. Section III contains background information on the theory and mathematics of four types of Kalman filters, along with design and coding information for one filter. Section IV describes the requirements for test inputs for Kalman filters and the manner in which these inputs are provided to the program which tests the Kalman filter. Evaluating the performance of a Kalman filter implementation is discussed in Section V, along with some typical results for one implementation. Section VI presents a summary of the experience gained in producing the various packages which form the Kalman filter program, describes additional work needed, and discusses advantages of producing reusable software components.

II. REUSABILITY ISSUES

The process of creating a Kalman filter program with its associated testing environment illustrates on a small scale what can be accomplished on a larger scaled software project.

The generic Kalman filter packages discussed in this paper represent part of a library of reusable Ada software components. Included among these components are packages providing general purpose mathematical routines, including trigonometric, exponential, and logarithmic functions, square root, and angle conversions [22]; conventional matrix and vector operations [23] to [27]; statistical operations [28] to [30]; and the capability of producing a graph of operational results.

Since the Kalman filter implementation requires matrix multiplication, it utilizes the generic matrix/vector operations package from the reusable parts library. This matrix/vector operations package requires a square root algorithm, which is provided in a generic math library package from the parts library. In order to test the Kalman filter package, the generic package KALMAN_FILTER_TEST was created.

A package of statistical operations which can be used to provide performance characteristics for the various Kalman filter implementations discussed here is being developed. Package STATISTICS_OPERATIONS will also become a reusable component in the parts library.

A test driver, procedure TEST_FILTER, was developed to execute the testing routines of KALMAN_FILTER_TEST. This interactive, menu-driven procedure required the facilities of the language-defined package, TEXT_IO, for input from the user, output to the screen, and writing test results to a file. During the design of TEST_FILTER, it was decided that the ability to graph the error of the Kalman filter would be a desirable feature. A package called DISPLAY_RESULTS was therefore produced and placed in the reusable parts library. It provides an interface to a graphics utility and has the option of producing a line graph, bar chart, or pie chart of the test results.

The dependencies among program units in Ada determine a partial ordering for compilation and recompilation. As shown in the structure chart in Figure 1, GENERIC_MATH_LIB, DISPLAY_RESULTS, and STATISTICS_OPERATIONS have no dependencies. The only compilation order requirement is that they must be compiled and placed in the library before the units which list them in a context clause. MATRIX_VECTOR_OPERATIONS depends on GENERIC_MATH_LIB and must be compiled after it. Figure 1 shows that KALMAN_FILTER must be compiled after MATRIX_VECTOR_OPERATIONS, and that the compilation of KALMAN_FILTER_TEST must follow that of KALMAN_FILTER and STATISTICS_OPERATIONS. TEST_FILTER must be compiled after KALMAN_FILTER_TEST and DISPLAY_RESULTS.

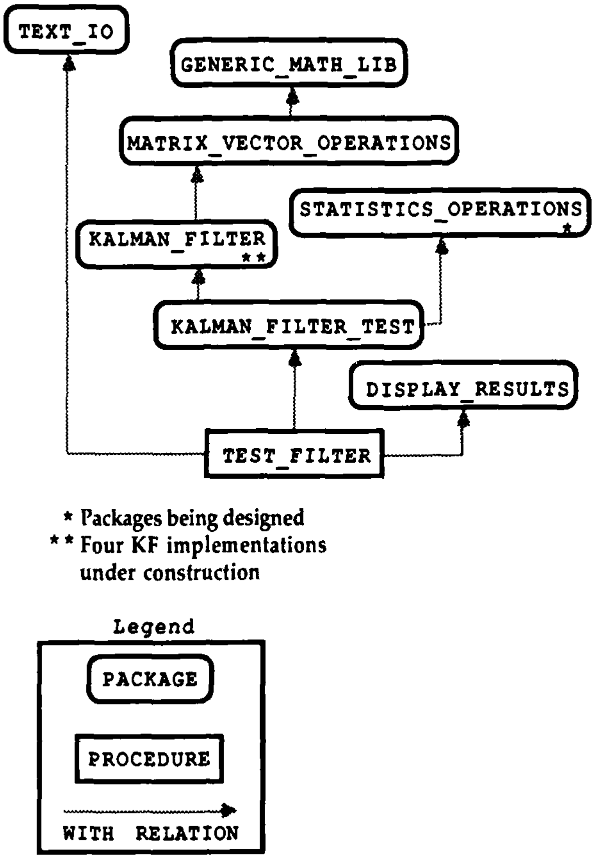


Figure 1. Structure of Ada Kalman Filter Program

Figure 1 shows the structure and dependencies of the Kalman filter package together with its interactive testing facility using symbology introduced by Burkhardt and Lee [31]. Five of the program units are reusable Ada software components, i.e., GENERIC_MATH_LIB, MATRIX_VECTOR_OPERATIONS, DISPLAY_RESULTS, STATISTICS_OPERATIONS, and KALMAN_FILTER. The I/O facilities are provided by the Ada language. The two remaining components, KALMAN_FILTER_TEST and TEST_FILTER, are application-specific units developed to test Kalman filter packages, although they contain Ada subprograms which may be easily adapted to other uses.

The Kalman filter program was constructed by combining language-defined Ada packages, existing reusable software components, and application-specific components. This method will be applicable to larger scale projects as the availability and number of reusable Ada components increase.

Ada's packaging mechanism provides a means of controlling those items available to the outside user. The specification of the Kalman filter package, as shown in Table 1, contains the instantiation of the generic MATRIX_VECTOR_OPERATIONS package with the same precision as KALMAN_FILTER. This package is placed in the specification to make it visible to any other unit needing the facilities of MATRIX_VECTOR_OPERATIONS. The packaging facility ensures that those items, and only those items, declared in the specification are available to the user of this package. However, neither items declared in the package body, nor the implementation details of the package, can be accessed by the user.

This feature is used to protect the state information stored between iterations of the Kalman filter. As long as the package is in scope, i.e., available for use, the state variables in the Kalman filter package body, unlike local objects in an Ada subprogram, will remain available and valid, retaining their values from iteration to iteration.

Experience in designing reusable parts has revealed few problems with writing generics instantiated within other generics. For example, in the Kalman filter, generic package GENERIC_MATH_LIB is instantiated in generic package MATRIX_VECTOR_OPERATIONS, which is then instantiated within generic package KALMAN_FILTER, which is then instantiated within generic package KALMAN_FILTER_TEST. KALMAN_FILTER_TEST is instantiated within procedure TEST_FILTER, the main procedure. By using levels of generics in this way, all testing is performed in the precision with which TEST_FILTER instantiates KALMAN_FILTER_TEST. Table I contains code segments showing the instantiations.

Table I. Instantiations of Generic Packages

```
generic
  type YOUR_FLOAT is digits <>;
package GENERIC_MATH_LIB is
  ...
end GENERIC_MATH_LIB;
                 ************************
with GENERIC_MATH_LIB;
generic
  type YOUR_FLOAT is digits <>;
package MATRIX_VECTOR_OPERATIONS is
  package MY_MATH_LIB is new GENERIC_MATH_LIB (YOUR_FLOAT);
  use MY_MATH_LIB;
  ...
end MATRIX_VECTOR_OPERATIONS;
                 **********************
with MATRIX_VECTOR_OPERATIONS;
generic
  type YOUR_FLOAT is digits <>;
  ...
package KALMAN_FILTER is
  package MY_MATRIX_VECTOR_OPERATIONS is new
          MATRIX_VECTOR_OPERATIONS (YOUR_FLOAT);
  use MY_MATRIX_VECTOR_OPERATIONS;
  ...
end KALMAN_FILTER;
                 **********************
with KALMAN_FILTER;
generic
  type YOUR_FLOAT is digits <>;
package KALMAN_FILTER_TEST is
  package MY_KALMAN_FILTER is new KALMAN_FILTER (YOUR_FLOAT, 9, 6);
  use MY_KALMAN_FILTER;
  use MY_MATRIX_VECTOR_OPERATIONS;  --instantiated in KALMAN_FILTER.
  ...
end KALMAN_FILTER_TEST;
                 ***********************
with KALMAN_FILTER_TEST;
...
procedure TEST_FILTER is
  type TEST_FLOAT is digits 6;
  ...
  package MY_KALMAN_FILTER_TEST is new
          KALMAN_FILTER_TEST (TEST_FLOAT);
  use MY_KALMAN_FILTER_TEST;
  use MY_KALMAN_FILTER;          --instantiated in KALMAN_FILTER_TEST.
  use MY_MATRIX_VECTOR_OPERATIONS; --instantiated in KALMAN_FILTER.
  ...
end TEST_FILTER;
```

A problem can occur, however, if the structure is not designed carefully. In the example above, both KALMAN_FILTER and KALMAN_FILTER_TEST require access to package MATRIX_VECTOR_OPERATIONS. If the package KALMAN_FILTER_TEST instantiates MATRIX_VECTOR_OPERATIONS and KALMAN_FILTER with the type specified by its own generic parameter, and the package KALMAN_FILTER instantiates package MATRIX_VECTOR_OPERATIONS with its generic parameter, then KALMAN_FILTER_TEST and KALMAN_FILTER would each contain a version of MATRIX_VECTOR_OPERATIONS instantiated with their own generic parameters. However, since the generic parameter of KALMAN_FILTER is the same as the generic parameter for KALMAN_FILTER_TEST, one might assume that the two instantiations of MATRIX_VECTOR_OPERATIONS would be compatible, though this assumption is not necessarily valid. Suppose the package MATRIX_VECTOR_OPERATIONS declares types MATRIX and VECTOR, which are based upon its generic parameter. If the package KALMAN_FILTER_TEST declares objects of type MATRIX and attempts to pass those objects to a subprogram of KALMAN_FILTER which expects objects of type MATRIX, a particular compiler may not accept those calls and may complain that the call cannot be resolved. This is due to the fact that KALMAN_FILTER_TEST and KALMAN_FILTER instantiated MATRIX_VECTOR_OPERATIONS with their own generic parameters. Although the parameters are identical, a compiler may consider them distinct and therefore not allow them to be used as though they were the same.

A solution to this potential problem with generics is to perform instantiations of generic packages needed by package A in the specification of package A, rather than in its body, if those packages are needed by another unit which uses package A (directly or indirectly). That other unit would then use the instantiation in package A instead of performing a new instantiation itself. This also eliminates the overhead of multiple (identical) instantiations of generic packages within the same program, as well as resolving possible overloading ambiguities.

As observed by St. Dennis, Stachour, Frankowski, and Onuegbe [32], reusable software is built for reuse; is fit for reuse, i.e., is composable with other code without interfering with the other code, nor allowing itself to be interfered with; and presents a useful abstraction at an appropriate level of abstraction. The reusable components in the Kalman filter program and, in general, all the components in the reusable parts library, are primarily designed for reuse. Others are developed as a necessary or desirable feature for a specific application, and only after development is the part itself recognized as an appropriate candidate for the reusable library. In this case, the part is modified, perhaps even redesigned, to become a reusable component.

In the development of the reusable Ada software components, it was realized early on that an overwhelming concern must be given to readability and understandability by the user. To this end, an internal Ada Style Guide was developed to guide the construction of the reusable parts. Methods of formatting text in order to enhance readability are included in the style guide. Naming conventions are established to make the source code more readable and self-documenting. Guidelines are established on the use of comments, exceptions, representation clauses, and other language features. Included are suggestions on program structuring and guidelines on appropriate use of different program units.

In addition to the Ada Style Guide, header templates in the form of comment blocks were created for each type of program unit to be developed. There are different templates for a subprogram compilation unit, an embedded subprogram, a package specification, and a package body. The header for a subprogram compilation unit includes information about the input, output, action performed, method used to perform the action, supporting routines and dependencies, exceptions, embedded subprograms, types and subtypes, and a data dictionary. For an embedded subprogram, the header adds a section for listing those units which call the subprogram. Package specification and body headers contain similar information to a subprogram compilation unit, but also include a list of all functions and procedures contained in the package.

Embedded comment blocks precede major/critical sections of code to ensure that the reader understands the purpose of those sections. These blocks are also used to clarify complicated mathematical operations.

Before a component is placed in the reusable Ada parts library it is reviewed by several members of the staff, in addition to the implementer, to ensure that the required style guidelines have been followed for the source code, that there are appropriate headers and sufficient internal comments, and that the testing method is sufficient to validate the code.

III. KALMAN FILTER PACKAGES

History of Kalman Filters

Kalman filters are named after R. E. Kalman, who presented his paper, "A New Approach to Linear Filtering and Prediction Problems", at the Instruments and Regulators Conference, March 29 to April 2, 1959, of THE AMERICAN SOCIETY OF MECHANICAL ENGINEERS. His 1960 paper [11] related his work to the prior work of Wiener [33] in 1949, Zadeh and Ragazzini [34] in 1950, and Bode and Shannon [35] in 1950. The Kalman filter concept found widespread applications and served as the basis for many new theoretical and computational developments.

In February 1970, the NATO Advisory Group for Aerospace Research and Development (AGARD) published a comprehensive set of reviews in its report, "Theory and Applications of Kalman Filtering" [36]. NATO AGARD published a second report in March 1982, "Advances in the Techniques and Technology of the Application of Nonlinear Filters and Kalman Filters" [37]. In March 1983, THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS devoted a special issue of its Transactions on Automatic Control to the subject, "Applications of Kalman Filtering". There are many other publications on KF which demonstrate its utility. With the advent of modern digital signal processing [38] to [42] and the application of statistical concepts to filtering [6], it has become practical to estimate the state of a linear system with noisy inputs using noisy measurements. In its simplest form, the Kalman filter algorithm yields: 1) an estimate of the system state (n x 1 state vector), and 2) a measure of the quality of the estimate (n x n error covariance matrix). At each discrete time, the KF algorithm produces a new state estimate and a new error covariance. Numerical difficulties with the simple KF have led to other KF implementations which resolve some of these difficulties.

In view of this KF history and the major role of KF in tracking and control systems, it is apparent that an Ada library [43] and [44] of reusable components must include a means to implement Kalman filters and to test their performance. Therefore, the objective has been set to establish generic Ada packages for several KF implementations [45] and for the necessary supporting packages and subprograms.

Selected Implementations

Four types of Kalman filters have been selected for inclusion in the Ada library of reusable components: the conventional Kalman filter (CKF), the square root covariance filter (SRCF), the Chandrasekhar square root filter (CSRF), and the square root information filter (SRIF). Verhaegen and Van Dooren [46] have treated these four KF algorithms, and their paper provides the necessary analysis and references. The reader should also consult [47] and [48] for additional KF theory and algorithm information.

In the CKF, the new measurement vector and previous values of the state estimate vector and of the error covariance matrix are combined at each discrete time to produce a new state estimate vector and a new error covariance matrix. This recursive process is called propagation. The propagation of the error covariance matrix can result in ill-conditioned quantities for particular measurement and system conditions. Potter [49] and [50] gave a method for propagation of the error covariance matrix in a square root form. Kaminski, Bryson, and Schmidt [48] have published a survey of algorithms used to propagate state estimate vectors and error covariance matrices in square root forms. The term square root, widely used in KF literature, is more properly called the Cholesky factor [51].

A different algorithm for performing the KF function, devised by Fraser [52], is the information filter. The information algorithm, discussed by Anderson and Moore [10], provides recursive filtering by propagating a reference vector and an information matrix. The system state estimate vector can be calculated (when needed) from the reference vector and the information matrix. The information matrix is essentially the Cholesky factor of the inverse of the error covariance matrix. Kailath [53] has shown that filtering algorithms may be based upon either the ancient (1724) Ricatti-type difference equation [54] or the Chandrasekhar-type equation introduced in 1948 for problems in astrophysics. These algorithms offer advantages and disadvantages from the point of view of numerical computations.

Theoretical developments by Andrews [55] have shown that the measurement vector can be applied one component at a time in a technique called sequential processing. When the entire measurement vector is used at once, it is called simultaneous processing. KF implementations are usually divided into two parts. In the first part, the previous state vector and covariance matrix are replaced by new predicted or extrapolated values. In the second part, the state vector and covariance matrix are adjusted to new values based upon the predicted values and new measurements. In some KF algorithms, simultaneous processing is used in the time update portion and sequential processing is used in the measurement update portion.

Many KF implementations are now available, based upon the choices of covariance/information matrix propagation, simultaneous/sequential processing, square root factorization, and Ricatti/Chandrasekhar equations. Additional choices result from the various kinds of matrix operations that are available. The reader is referred to many books on numerical methods [56] to [61]. In the following paragraphs of this section, the essential equations for each of the four selected types of KF implementations are given approximately as they would be expressed in Ada statements.

Matrix Mathematics

The package ADVANCED_MATRIX_OPERATIONS contains subprograms which accomplish Cholesky decomposition, Householder triangularization, modified Gram-Schmidt orthogonalization, a back-substitution solution for triangularized linear equations, and other sophisticated matrix operations.

The Cholesky decomposition of a symmetric, positive, semidefinite n x n matrix A, produces a lower triangular n x n matrix L which solves the matrix equation:

$$L * TRANSPOSE(L) = A \tag{1}$$

The call to this function passes matrix A:

$$L := CHOLESKY\_FACTOR(A); \tag{2}$$

The Householder triangularization of an (n+m) x n partitioned matrix A produces an n x n upper triangular matrix W, using implicitly an (n+m) x (n+m) orthogonal transformation matrix T. When A is a 2 x 1 partitioned matrix, with matrix elements A1 and A2, the procedure HOUSEHOLDER_2X1 solves the equation below for matrix W:

$$\begin{array}{r|c|} n & W \\ \hline m & 0 \\ & n \end{array} = T \begin{array}{|c|l} A1 & n \\ \hline A2 & m \\ n & \end{array} \tag{3}$$

The call to this procedure names the inputs A1 and A2 and the output W:

$$HOUSEHOLDER\_2X1(A1,A2,W); \tag{4}$$

When A is a 2 x 2 partitioned matrix, procedure HOUSEHOLDER_2X2 solves the matrix equation below for scalar F, row vector G, and matrix W:

$$\begin{array}{r|c|c|} 1 & F & G \\ \hline n & 0 & W \\ & 1 & n \end{array} = T \begin{array}{|c|c|l} A1 & 0 & 1 \\ \hline A2 & A3 & n \\ 1 & n & \end{array} \tag{5}$$

The call to this procedure names the inputs (scalar A1, column vector A2, matrix A3) and the outputs (F, G, and W) :

$$HOUSEHOLDER\_2X2(A1,A2,A3,F,G,W); \tag{6}$$

The modified Gram Schmidt orthogonalization algorithm also solves matrix equations (3) and (5) with the calls

$$MOD\_G\_S\_2X1(A1,A2,W); \tag{7}$$

$$MOD\_G\_S\_2X2(A1,A2,A3,F,G,W); \tag{8}$$

When U is an n x n upper triangular matrix and B is a constant vector, a set of linear equations for vector Z is:

$$\begin{array}{r} U11\,Z1 + U12\,Z2 + \ldots + U1n\,Zn = B1 \\ U22\,Z2 + \ldots + U2n\,Zn = B2 \\ Unn\,Zn = Bn \end{array} \tag{9}$$

These equations are easily solved by a back-substitution algorithm which starts with the last equation for Zn. The call for procedure BACK_SUBSTITUTION names the inputs (matrix U and column vector B) and the output (vector Z):

$$BACK\_SUBSTITUTION(U,B,Z); \tag{10}$$

Conventional Kalman Filter

Conventional Kalman filter (CKF) mathematics are presented and explained in this paragraph. The linear system and observations in this paper are modeled according to the following equations:

```
- - PROCESS EQUATIONS (n-component vector)
STATE_NEW := STATE_TRANSITION * STATE_OLD +
   PROCESS_NOISE_CONVERSION * PROCESS_NOISE;
- - OBSERVATION EQUATIONS (r-component vector)
MEASUREMENT := MEASURE_CONVERSION * STATE_OLD + MEASURE_NOISE;
```

The covariances of PROCESS_NOISE and MEASURE_NOISE are known and are represented by COV_PROCESS_NOISE and COV_MEASURE_NOISE. The matrices STATE_TRANSITION, PROCESS_NOISE_CONVERSION, and MEASURE_CONVERSION are also known and may be time-varying.

The CKF algorithm treated in this paper consists of two parts: 1) the time update of the state vector and the error covariance matrix using values of this vector and matrix from the previous iteration; and 2) measurement update of the state vector and error covariance matrix, using a calculated gain matrix and the vector of measurements. The equations used for these CKF calculations are as follows:

```
- - TIME UPDATE
STATE_EXTRAPOLATION := STATE_TRANSITION * STATE_OLD;
ERROR_EXTRAPOLATION := STATE_TRANSITION * ERROR_OLD * TRANSPOSE
   (STATE_TRANSITION) + PROCESS_NOISE_CONVERSION *
   COV_PROCESS_NOISE * TRANSPOSE(PROCESS_NOISE_CONVERSION);
- - MEASUREMENT UPDATE
GAIN := ERROR_EXTRAPOLATION * TRANSPOSE(MEASURE_CONVERSION) *
   INVERSE(MEASURE_CONVERSION * ERROR_EXTRAPOLATION *
   TRANSPOSE(MEASURE_CONVERSION) + COV_MEASURE_NOISE);
STATE_NEW := STATE_EXTRAPOLATION + GAIN * (MEASUREMENT -
   MEASURE_CONVERSION * STATE_EXTRAPOLATION);
ERROR_NEW := ERROR_EXTRAPOLATION - GAIN * MEASURE_CONVERSION *
   ERROR_EXTRAPOLATION;
- - END OF UPDATES
```

These equations require the use of simple matrix and vector operations available from the reusable package MATRIX_VECTOR_OPERATIONS. The ERROR_NEW matrix can become ill-conditioned as a result of numerical problems, and the GAIN matrix can be a problem due to the matrix inversion operation. If the GAIN becomes small, the measurements are essentially ignored in the calculation of STATE_NEW. If the GAIN becomes large, measured values become dominant.

Square Root Covariance Filter

In the square root covariance filter (SRCF), the error covariance matrix of the conventional Kalman filter (CKF) is replaced with a lower triangular Cholesky factor [61] (square root) of the error covariance matrix. The product of the Cholesky factor and its transpose is the error covariance matrix; this product can never fail to be non-negative-definite. Its conditioning is better than that of the error covariance matrix. The Cholesky factor yields a double precision effect in numerical computations.

The SRCF statements below are based on the equations given by Kaminski, Bryson, and Schmidt [48]. The time update is done in the simultaneous processing mode with a single vector operation. The measurement update is done in the sequential processing mode by means of a loop in which one scalar component of the measurement vector is processed in each iteration. If the covariance matrix of the measurement noise is not diagonal, an additional Cholesky decomposition is necessary to make it diagonal ([48] pp. 730-731).

```
- - TIME UPDATE
STATE_EXTRAPOLATION:= STATE_TRANSITION * STATE_OLD;
ROOT_PROCESS_NOISE := CHOLESKY_FACTOR(COV_PROCESS_NOISE);
TEMP_MATRIX_1 := TRANSPOSE(ROOT_OLD) * TRANSPOSE(STATE_TRANSITION);
TEMP_MATRIX_2 := TRANSPOSE(ROOT_PROCESS_NOISE) *
   TRANSPOSE(PROCESS_NOISE_CONVERSION);
HOUSEHOLDER_2X1(TEMP_MATRIX_1,TEMP_MATRIX_2,OUTPUT_MATRIX);
ROOT_EXTRAPOLATION := TRANSPOSE(OUTPUT_MATRIX);
- - MEASUREMENT UPDATE
ROOT_MEASURE_NOISE := CHOLESKY_FACTOR(COV_MEASURE_NOISE);
for I in 1..OBSERVATION_SIZE loop
TEMP_SCALAR := ROOT_MEASURE_NOISE(I,I);
TEMP_VECTOR := TRANSPOSE(ROOT_EXTRAPOLATION)
   * COLUMN(I,TRANSPOSE(MEASURE_CONVERSION));
TEMP_MATRIX := TRANSPOSE(ROOT_EXTRAPOLATION);
HOUSEHOLDER_2X2(TEMP_SCALAR, TEMP_VECTOR, TEMP_MATRIX,
   NEW_SCALAR, NEW_VECTOR,NEW_MATRIX);
FACTOR := NEW_SCALAR;
GAIN := NEW_VECTOR;
STATE_NEW := (1.0/FACTOR) * TRANSPOSE(GAIN) * (MEASUREMENT -
   MEASURE_CONVERSION * STATE_EXTRAPOLATION) + STATE_OLD;
ROOT_NEW := TRANSPOSE(NEW_MATRIX);
end loop;
- - END OF UPDATES
```

The implementation of the SRCF requires matrix operations for Cholesky decomposition, Householder triangularization, modified Gram-Schmidt orthogonalization, back-substitution solution of a linear equation with a triangular coefficient matrix, and selection of a column vector from a matrix.

Chandrasekhar Square Root Filter

The mathematical basis for the CKF, SRCF, and SRIF are process and observation equations with time-varying matrices: state-transition, measurement-conversion, process-noise-conversion, process-noise-covariance, and measurement-noise-covariance. The error optimization calculations for those filters make use of a nonlinear, first-order differential equation treated by Ricatti in 1724 [54]. In 1973, Kailath [53] showed that, when these matrices are time-invariant, optimization could be based upon a differential equation introduced in 1948 by the astrophysicist Chandrasekhar. Hence, for the time-invariant case, the Kalman filter equations can be replaced by equations for the Chandrasekhar square root filter (CSRF). The CSRF is considered to be a simplification of the SRCF achieved by making use of the time-invariance of these matrices. In the CSRF algorithm, the state estimate vector is propagated using the measurement vector and three auxiliary matrices. The error covariance matrix is propagated using one of the auxiliary matrices, which is a Cholesky factor of the increment of the error covariance matrix. The following Ada statements, based on [46] and [63], accomplish one recursive iteration of the CSRF.

```
- - COMBINED TIME AND MEASUREMENT UPDATES
HOUSEHOLDER_CSRF(ROOT_MEASURE_OLD, ROOT_ERROR_OLD, GAIN_OLD,
   STATE_TRANSITION, MEASURE_CONVERSION,
   ROOT_MEASURE_NEW, ROOT_ERROR_NEW, GAIN_NEW);
TEMP_MATRIX_1 := GAIN_NEW * INVERSE(CHOLESKY_FACTOR
   (COV_MEASURE_NOISE));
TEMP_MATRIX_2 := MEASUREMENT - MEASUREMENT_CONVERSION
   * STATE_OLD;
STATE_NEW := STATE_TRANSITION * STATE_OLD
   + TEMP_MATRIX_1 * TEMP_MATRIX_2;
ERROR_NEW := ERROR_OLD+ROOT_ERROR_OLD * TRANSPOSE(ROOT_ERROR_OLD);
GAIN_OLD := GAIN_NEW;
ROOT_MEASURE_OLD := ROOT_MEASURE_NEW;
ROOT_ERROR_OLD := ROOT_ERROR_NEW;
STATE_OLD := STATE_NEW;
ERROR_OLD := ERROR_NEW;
- - END OF UPDATES
```

The implementation of the above CSRF algorithm utilizes special skew Householder transformations provided in the package ADVANCED_ MATRIX_OPERATIONS. The numerical properties of these non-Ricatti-based estimations have been reported in the literature cited in [62] and [63].

Square Root Information Filter

The "information" matrix is defined to be the inverse of the error covariance matrix. The Cholesky decomposition algorithm, when applied to the information matrix, yields a lower triangular matrix called the square root information matrix. Use of matrix mathematics shows that the propagation of the state estimate vector and the error covariance matrix, as in the CKF, can be replaced by the propagation of the information vector and the square root information matrix. The algorithm which achieves this propagation is called the square root information filter (SRIF). Described in the published literature [10], [19], [46], and [64], are SRIF algorithms which offer choices between simultaneous processing and sequential processing modes.

The statements below represent an algorithm from [46] which executes the measurement and time updates simultaneously. It is assumed that the covariance of the process noise is a diagonal matrix or that it has been made so by an appropriate Cholesky decomposition. The state estimate vector and the error covariance matrix can be obtained, when needed, from the variables produced in the SRIF.

```
- - MEASUREMENT AND TIME UPDATES
TEMP_MATRIX_1 := TRANSPOSE(CHOLESKY_FACTOR(INVERSE
   (COV_PROCESS_NOISE)));
TEMP_MATRIX_2 := ROOT_INFORM_OLD * INVERSE(STATE_TRANSITION);
TEMP_MATRIX_3 := TEMP_MATRIX_2 * PROCESS_NOISE_DISTRIBUTION;
TEMP_MATRIX_6 := TRANSPOSE(CHOLESKY_FACTOR(INVERSE
   (COV_MEASURE_NOISE)));
TEMP_MATRIX_4 := TEMP_MATRIX_6 * MEASURE_CONVERSION;
TEMP_MATRIX_5 := TEMP_MATRIX_6 * MEASUREMENT;
HOUSEHOLDER_3X3(TEMP_MATRIX_1, TEMP_MATRIX_2, TEMP_MATRIX_3,
   TEMP_MATRIX_4,TEMP_MATRIX_5, INFORM_VECTOR_OLD,
   INFORM_VECTOR_NEW, ROOT_INFORM_NEW);
INFORM_VECTOR_OLD := INFORM_VECTOR_NEW;
ROOT_INFORM_OLD := ROOT_INFORM_NEW;
- - END UPDATES
```

The above implementation of the SRIF algorithm makes use of special versions of the Householder triangularization algorithm. HOUSEHOLDER_3X3 is provided in the package ADVANCED_ MATRIX_OPERATIONS. The SRIF offers numerical advantages which have led to its wide use.

Filter Package Design

An Ada package consists of two parts: the specification and the body. The specification may be compiled separately, but must be compiled before the body is compiled. The specification provides the outside interface. The body contains the hidden details, package initialization statements, and exception handlers. Those procedures, functions, and variables intended to be visible outside of the package are controlled by the specification. The use of the Ada "generic" mechanism permits the parameterization of packages with types, values, and objects.

These Ada features have been used effectively in designing Kalman filter packages. The procedures for initializing and running the filter are made visible to other parts of the program by suitable declarations in the package specification. The generic feature has been used to permit the external control of the numerical precision (number of decimal digits) used. Variables given visibility include scalars, vectors, and matrices. The state estimation vector and error covariance matrix, for example, are declared and made visible in the specification. The detailed statements needed for each KF algorithm, as well as local variables, are hidden in the package body. The Ada "with" and "use" constructs make available for use within the KF package, those mathematical, vector, and matrix operations provided in other packages. The exception handling features of the Ada language are used within the package body in techniques which deal with ill-conditioned matrices and other numerical problems recognized during the performance of the internal KF operations. The Ada package construct has been found to be a convenient and useful means for treating each specific KF implementation.

Example : CKF Package

Ada implementation of the CKF was designed as a generic package with three parameters: the precision of the floating point type, the size of the state vector, and the size of the observation vector. The package consists of a procedure to initialize the filter, a procedure to run the filter, a type used for reliability, and variables which store values of the state estimate and error covariance produced during the previous run of the filter. The source code for the specification and body of the generic package KALMAN_FILTER is given in Table II. To conserve space in this paper, extensive documentation has been removed from this listing. The usual code format, which stresses readability, has also been sacrificed in some places for the same reason.

Table II. Source Code for Package KALMAN_FILTER_CKF

```
-- COPYRIGHT (c) 1987 Martin Marietta Corp. All rights reserved.

with MATRIX_VECTOR_OPERATIONS;

generic
  type YOUR_FLOAT is digits <>;
  STATE_SIZE        : POSITIVE;
  OBSERVATION_SIZE : POSITIVE;

package KALMAN_FILTER_CKF is

  package MY_MATRIX_VECTOR_OPERATIONS is new
          MATRIX_VECTOR_OPERATIONS (YOUR_FLOAT);
  use MY_MATRIX_VECTOR_OPERATIONS;
  type RELIABILITY is (GOOD, QUESTIONABLE, BAD);

  procedure INITIALIZE_KALMAN_FILTER
                  (INITIAL_STATE_ESTIMATE : in VECTOR;
                   INITIAL_ERROR_COV      : in MATRIX);
  procedure RUN_KALMAN_FILTER
                  (OBSERVATION_VALUE      : in     VECTOR;
                   STATE_TRANSITION       : in     MATRIX;
                   OBSERVATION_CONVERSION : in     MATRIX;
                   OBSERVATION_NOISE_COV  : in     MATRIX;
                   STATE_NOISE_COV        : in     MATRIX;
                   STATE_ESTIMATE_NEW     :    out VECTOR;
                   ERROR_COV_NEW          :    out MATRIX;
                   KF_RELIABILITY         :    out RELIABILITY);

end KALMAN_FILTER_CKF;


package body KALMAN_FILTER_CKF is

  ERROR_COV_OLD      : MATRIX (1 .. STATE_SIZE, 1 .. STATE_SIZE)
                       := (others => (others => 0.0));
  STATE_ESTIMATE_OLD : VECTOR (1 .. STATE_SIZE) := (others => 0.0);

  procedure INITIALIZE_KALMAN_FILTER
                      (INITIAL_STATE_ESTIMATE : in VECTOR;
                       INITIAL_ERROR_COV      : in MATRIX) is
  begin

    ERROR_COV_OLD      := INITIAL_ERROR_COV;
    STATE_ESTIMATE_OLD := INITIAL_STATE_ESTIMATE;

  end INITIALIZE_KALMAN_FILTER;

  procedure RUN_KALMAN_FILTER
                 (OBSERVATION_VALUE      : in     VECTOR;
                  STATE_TRANSITION       : in     MATRIX;
                  OBSERVATION_CONVERSION : in     MATRIX;
                  OBSEFVATION_NOISE_COV  : in     MATRIX;
                  STATE_NOISE_COV        : in     MATRIX;
                  STATE_ESTIMATE_NEW     :    out VECTOR;
                  ERROR_COV_NEW          :    out MATRIX;
                  KF_RELIABILITY         :    out RELIABILITY) is

    LOCAL_STATE_ESTIMATE_NEW : VECTOR (1 .. STATE_SIZE);
    LOCAL_ERROR_COV_NEW : MATRIX (1..STATE_SIZE, 1..STATE_SIZE);
    GAIN : MATRIX (1 .. STATE_SIZE, 1 .. OBSERVATION_SIZE);
    INVERSE_MATRIX:MATRIX(1..OBSERVATION_SIZE,1..OBSERVATION_SIZE);
    ILL_CONDITIONED_MATRIX : BOOLEAN;

  begin  -- RUN_KALMAN_FILTER
    LOCAL_STATE_ESTIMATE_NEW := STATE_TRANSITION *
       STATE_ESTIMATE_OLD;
    LOCAL_ERROR_COV_NEW := (STATE_TRANSITION * ERROR_COV_OLD *
       (TRANSPOSE (STATE_TRANSITION))) + STATE_NOISE_COV;
    COMPUTE_INVERSE ((OBSERVATION_CONVERSION * LOCAL_ERROR_COV_NEW *
       (TRANSPOSE(OBSERVATION_CONVERSION)))+OBSERVATION_NOISE_COV),
       INVERSE_MATRIX, ILL_CONDITIONED_MATRIX);
    GAIN := LOCAL_ERROR_COV_NEW *
       (TRANSPOSE (OBSERVATION_CONVERSION)) * INVERSE_MATRIX;
    LOCAL_ERROR_COV_NEW := LOCAL_ERROR_COV_NEW - (GAIN *
       OBSERVATION_CONVERSION * LOCAL_ERROR_COV_NEW);
    LOCAL_STATE_ESTIMATE_NEW := LOCAL_STATE_ESTIMATE_NEW + (GAIN *
       (OBSERVATION_VALUE - (OBSERVATION_CONVERSION *
       LOCAL_STATE_ESTIMATE_NEW)));

    ERROR_COV_OLD      := LOCAL_ERROR_COV_NEW;
    STATE_ESTIMATE_OLD := LOCAL_STATE_ESTIMATE_NEW;

    ERROR_COV_NEW      := LOCAL_ERROR_COV_NEW;
    STATE_ESTIMATE_NEW := LOCAL_STATE_ESTIMATE_NEW;

    if ILL_CONDITIONED_MATRIX then
      KF_RELIABILITY    := QUESTIONABLE;
    else
      KF_RELIABILITY    := GOOD;
    end if;

  exception
    when others => KF_RELIABILITY := BAD;
  end RUN_KALMAN_FILTER;

end KALMAN_FILTER_CKF;

-- COPYRIGHT (c) 1987 Martin Marietta Corp. All rights reserved.
```

IV. TESTING METHODS

Before an implementation of a Kalman filter can be stored in a library of reusable components, the code must be thoroughly tested to ensure that the filter performs correctly. To test the filter, a procedure which serves as a user interface and a package of simulation procedures were created. After the user of the testing procedure sets the parameters for a run of the test, the filter is initialized. A simulated target is produced, Gaussian distributed noise is added to target state variables, and this observation vector is passed with the user-specified parameters in a call to the Kalman filter procedure. The Kalman filter produces a state estimate, an error covariance matrix, and a reliability measure based upon its arguments. The state estimate and error covariance matrix are returned by the filter, and are also stored in the Kalman filter package variables for use in the next call to the filter. Following completion of a filter iteration, information about the accuracy of the state estimate is stored for later analysis. A file is also generated with data for use in graphing the accuracy of the filter, and a third file stores values of variables for use in tracking down problems with the filter, if the debugging mode is turned on. A typical cycle of testing the Kalman filter is given in Figure 2.

Package KALMAN_FILTER_TEST is a generic package which provides the subprograms needed to test the filter in a Kalman filter package. This test package has one generic parameter used to establish test precision. The package instantiates the generic Kalman filter package with the same precision, and with state size six and observation size nine. The package specification contains the variables which may be modified interactively by the user during testing and contains specifications for three procedures: INITIALIZE_FILTER, SET_TO_DEFAULT_VALUES, and PERFORM_TEST. The body contains variables used in generating Gaussian-distributed random noise and contains the three procedures listed above and procedures GENERATE_NOISE, SIMULATE_TARGET, and PRODUCE_OBSERVATIONS.

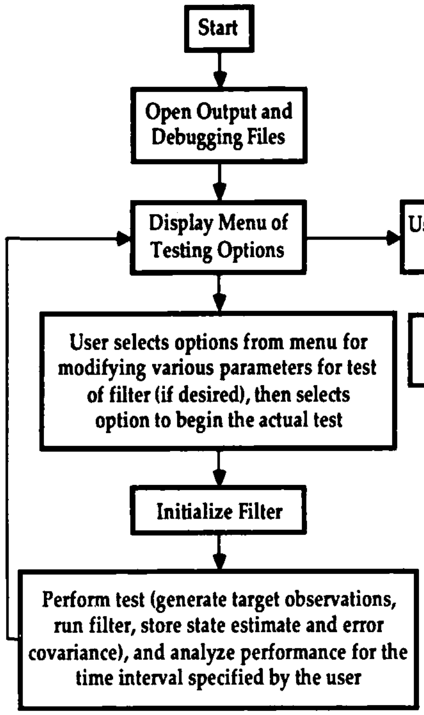

Start
Open Output and Debugging Files
Display Menu of Testing Options
User Selected Option to End Testing
User selects options from menu for modifying various parameters for test of filter (if desired), then selects option to begin the actual test
Close Output and Debugging Files
Stop
Initialize Filter
Perform test (generate target observations, run filter, store state estimate and error covariance), and analyze performance for the time interval specified by the user

Figure 2. Typical Cycle of Kalman Filter Testing

Procedure INITIALIZE_FILTER initializes package matrices not directly user-modifiable from the testing procedure. The procedure then calls procedure INITIALIZE_KALMAN_FILTER in the Kalman filter package with the user-provided (or default) initial state estimate and initial error covariance which will be used during the first iteration of the filter. Procedure SET_TO_DEFAULT_VALUES initializes to their default values all variables which may be modified by the user.

Procedure SIMULATE_TARGET computes the state vector for a simulated target moving in a rectangular, Cartesian, inertial reference frame with X-axis to NORTH, Y-axis to EAST, and Z-axis down (negative Z-axis is up). The target moves on a spiral three-dimensional path, with separate sine-wave modulation superimposed on radius, height, and angle coordinates.

Procedure GENERATE_NOISE produces six independent, uncorrelated, Gaussian-distributed, zero mean, random variables. This process utilizes two embedded subprograms, PRODUCE_UNIFORM_VALUE and PRODUCE_GAUSS_VALUES. Two values uniformly distributed between zero and one are generated, a random variable with Rayleigh probability distribution is generated, and then two independent Gaussian random variables are generated [65]. This process is repeated until all six random variables are created.

Procedure PRODUCE_OBSERVATIONS calls the procedures listed above to acquire a state vector based on the simulated target and a vector of noise. The noise is then modified to reflect desired standard deviations for position and for velocity. The procedure then combines the state and noise vectors into an observation vector, which it returns. This procedure is shown without comments in Table III.

Table III. Portions of Code From Package KALMAN_FILTER_TEST

```
package body KALMAN_FILTER_TEST is

  use MY_MATH_LIB;  -- instantiated in MATRIX_VECTOR_OPERATIONS

  -- ***** SEVERAL PORTIONS OF CODE REMOVED TO CONSERVE SPACE *****

  procedure PRODUCE_OBSERVATIONS is
  begin

    SIMULATE_TARGET;
    GENERATE_NOISE (GAUSSIAN_NOISE);

    -- LOOP TO MODIFY GAUSSIAN NUMBERS TO REFLECT STANDARD
    -- DEVIATIONS REMOVED TO CONSERVE SPACE IN LISTING

    OBSERVATION_MEASUREMENT := TRUE_STATE_MEASUREMENT (1 .. 6) +
                               GAUSSIAN_NOISE;

  end PRODUCE_OBSERVATIONS;


  procedure PERFORM_TEST is
  begin

    PRODUCE_OBSERVATIONS;
    GET_CPU_TIME (START_TIME);
    RUN_KALMAN_FILTER (OBSERVATION_MEASUREMENT,
        STATE_TRANSITION_SET, OBSERVATION_CONVERSION_SET,
        OBSERVATION_NOISE_COV_SET, STATE_NOISE_COV_SET,
        STATE_ESTIMATE, ERROR_COVARIANCE, RELIABILITY_VALUE);
    GET_CPU_TIME (STOP_TIME);
    DELTA_CPU_TIME := STOP_TIME - START_TIME;

  end PERFORM_TEST;

begin    -- KALMAN_FILTER_TEST

  SET_TO_DEFAULT_VALUES;
  INITIALIZE_FILTER;

end KALMAN_FILTER_TEST;
```

Finally, as shown in Table III, procedure PERFORM_TEST causes one iteration of the filter procedure to be executed. After calling procedure PRODUCE_OBSERVATIONS to produce an observation vector, procedure PERFORM_TEST calls RUN_FILTER in the Kalman filter package to execute one run of the Kalman filter. Thus, calling PERFORM_TEST results in a simulated target and a set of random numbers being calculated, the two sets of values being combined to form an observation vector, and one run of the Kalman filter being executed. In addition, CPU time used by the procedure RUN_FILTER is recorded.

Procedure TEST_FILTER runs a test of package KALMAN_FILTER using procedures in package KALMAN_FILTER_TEST and user-supplied interactive parameters. Results of the test are reported to an output file. Several intermediate results are also reported to a debugging file if the user specifies that debugging should be turned on. This procedure uses package TEXT_IO and instantiates several input/output generic packages and generic package KALMAN_FILTER_TEST.

TEST_FILTER declares the floating point type to be used to instantiate all generic packages, then instantiates all necessary packages not already visible. After opening files for output and debugging information, the procedure displays the values of user-modifiable test parameters and a menu of options available to the user. These options include changing any modifiable parameters, toggling debugging on/off, resetting all modifiable parameters to their original default values, and running the test based upon the current set of parameters. The user may continue modifying the parameters and running tests as long as desired. When all desired tests are completed, the user chooses the option to end the test (or series of tests).

If the user selects an option to change a parameter or set of parameters, this is also done interactively, with instructions to the user and the old values of the parameters provided by the test driver appearing on the screen. When the user indicates that the test should be run with the current set of parameters, a message is sent to the screen, local variables are initialized, and the Kalman filter and testing package are initialized with a call to KALMAN_FILTER_TEST.INITIALIZE_FILTER. For each value in the user-specified (or default) set of times, KALMAN_FILTER_TEST.PERFORM_TEST is executed. A local procedure then records the results and accuracy of that iteration of the test, and any debugging information is output to the debugging file.

Following the completion of the entire set of time values, the test results are sent to the output file, and a message that the test has been completed is sent to the screen. The user is then returned to the main menu and may choose whether to continue testing or end the series of tests.

After at least one test of the filter has been run, the user also has the option to graph the results of the previous test, if the proper hardware and software are available. This allows the user to generate a line graph or bar chart of the mean square errors of the state estimate. An example of the main menu produced by procedure TEST_FILTER is shown in Table IV.

Table IV. Sample of Main Menu Used for Testing Kalman Filter

```
KALMAN FILTER TEST PROCEDURE  (DEBUGGING is turned off)

Current values of modifiable parameters: Time interval is 1.00000E-01
Initial time is     0.00000E+00          Last time is    1.00000E+00
Jerk covariance is  1.00000E-04
Std.dev.(position) is 1.00000E-02   Std.dev.(velocity) is 1.00000E-01

          FREQUENCY      PHASE ANGLE    AMPLITUDE      MEAN VALUE
RADIUS    3.00000E-02    0.00000E+00    1.00000E-01     3.00000E+00
ANGLE     1.00000E-03    1.60000E+00    5.00000E-03     1.00000E-02
HEIGHT    2.00000E-02    8.00000E-01    1.50000E-02    -1.00000E-03

Spiral location is       5.00000E+00    6.00000E+00    -1.00000E+00

Options :  0.  End the test (or series of tests).
           1.  Run the test with the above values.
           2.  Reset to all original default values.
           3.  Turn on debugging mode.
           4.  Modify time parameters.
           5.  Modify jerk covariance, std. deviations
           6.  Modify spiral location.
           7.  View/modify initial state vector.
           8.  View/modify initial covariance matrix.
           9.  Modify radius information.
          10.  Modify angle information.
          11.  Modify height information.
          12.  Graph error from previous run.
```

Extensive, but not exhaustive, testing is done on the components considered for the reusable Ada parts library. The number of test cases run will vary with the complexity of the unit. For example, in testing some of the algorithms in MATRIX_ VECTOR_ OPERATIONS, 2500 iterations have been run at once to determine if singularity or ill-conditioning of a randomly generated matrix could be observed. Each reusable part in the library will be accompanied by documentation describing the testing procedure and reporting the results of the testing performed as a final validation of the code. A maximum error bound between the value achieved and the value expected is included in the documentation as appropriate. Since many reusable parts have an interactive testing package available with the components, the potential user is free to perform his/her own testing. The content of the formal documentation and testing procedures is expected to evolve as the reusable parts library matures.

V. PERFORMANCE EVALUATION

Evaluation of the conventional Kalman filter (CKF) is divided into two parts: 1) performance in state estimation, and 2) run-time characteristics in embedded computers. The performance in state estimation is evaluated by simulating noisy observations of target motions and system noise, and by comparing the state estimates from the CKF filter with the known true states of the target. For example, a graphic display of the errors in the estimates of target coordinates is constructed easily with the use of package DISPLAY_RESULTS. Numerical evaluations, although not the principal objective of the work reported here, include study of error covariances in state estimates, filter divergence, and stability. The reader is referred to many excellent papers [66] to [70] for the characteristics of the various KF implementations.

In addition to evaluating the state estimates produced by the Kalman filter, run-time conditions have also been examined. In particular, the size of the source and object code, the number of statements in the source code, and the amount of CPU time needed for one execution of the Kalman filter, i.e., one call to procedure KALMAN_FILTER.RUN_KALMAN_FILTER, have been gathered based on results of compiling the set of files and running the test.

Tables V and VI present the current results described above. Object size and source size are given in number of blocks, with 512 bytes/block. Note that for the generic packages, i.e., GENERIC_MATH_LIB, MATRIX_VECTOR_OPERATIONS, KALMAN_FILTER, and KALMAN_FILTER_TEST, the sizes represent the combination of the specification and body. These sizes are small because the packages are generic. The number of statements in the source code is the number of semicolons in the source files which are not part of a string or comment.

Table V. Kalman Filter Size Information

Module name	# blocks object module	# blocks source code	# of statements in source code total(spec+body)
GENERIC_MATH_LIB.OBJ;1	2	158	166
MATRIX_VECTOR_OPERATIONS.OBJ;1	2	290	336
KALMAN_FILTER_CKF.OBJ;1	2	63	52
KALMAN_FILTER_TEST.OBJ;1	3	118	193
DISPLAY_RESULTS.OBJ;1 (body)	3	5	6
DISPLAY_RESULTS .OBJ;1 (spec.)	2	5	2
TEST_FILTER.OBJ;1	280	221	744

Table VI. RUN_KALMAN_FILTER Execution Time

Number of digits of precision	CPU time (milliseconds)
6	13.712
16	596.820

Performance analysis of the Kalman filter is accomplished by Ada package DISPLAY_RESULTS and Ada procedure ANALYZE_PERFORMANCE. DISPLAY_RESULTS serves as an interface to a system graphics package and allows the user to see a graph representing the amount of mean square error in the most recent set of times for which the filter was run. For each component of the state vector, ANALYZE_PERFORMANCE calculates the square of the difference between the estimate produced by the Kalman filter and the known true value produced by KALMAN_FILTER_TEST.SIMULATE_TARGET.

Procedure ANALYZE_PERFORMANCE updates the mean square error and sends additional information to the output file for later use by the test operator. Portions of the code from procedure ANALYZE_PERFORMANCE are given in Table VII.

Table VII. Portions of Code for Analysis of Results

```
procedure ANALYZE_PERFORMANCE is
  ERROR : TEST_FLOAT;
begin

-- Write differences between true and estimated state vectors to
-- output file.  Actual code removed to conserve space.

-- Update the cumulative error term.

   for COUNTER in STATE_ESTIMATE'RANGE loop
     ERROR := TRUE_STATE_MEASUREMENT (COUNTER) -
              STATE_ESTIMATE (COUNTER);
     CUMULATED_FILTER_ERROR (COUNTER) :=
              (CUMULATED_FILTER_ERROR (COUNTER) *
              ((ITERATION_NUMBER - 1.0) / ITERATION_NUMBER)) +
              ((ERROR * ERROR) / ITERATION_NUMBER);
   end loop;

  end ANALYZE_PERFORMANCE;
```

The creation of the entire Kalman filter program, the components of which are shown in Figure 1, involved the use of some existing reusable Ada components from the parts library, the development of some components specific to this application, and the identification of some desirable components which are or eventually will become components in the reusable Ada parts library. As mentioned previously, GENERIC_MATH_LIB and MATRIX_VECTOR_OPERATIONS were existing components in the library at the beginning of the project. The goal of the project was to develop several implementations of Kalman filters to become reusable components. The problem of adequately testing the filters and viewing the results in some way, brought about the creation of new reusable components, such as DISPLAY_RESULTS and STATISTICS_OPERATIONS, as well as routines currently embedded within other program units which will soon be developed as independent reusable components.

Embedded within package KALMAN_FILTER_TEST are two such subprograms, GENERATE_NOISE and SIMULATE_TARGET. They were developed to simulate system noise and target motion in order to adequately and sufficiently test the Kalman filters. A more detailed description of these subprograms was given in Section III. Table VIII contains a list of the reusable components used in the Kalman filter program along with their current status.

Table VIII. Reusable Parts in Kalman Filter Program

UNIT NAME	UNIT STATUS
GENERIC_MATH_LIB	Completed/In Library.
MATRIX_VECTOR_OPERATIONS	Completed/In Library.
DISPLAY_RESULTS	Completed/In Library.
ADVANCED_MATRIX_OPERATIONS	Under development/Testing Phase.
STATISTICS_OPERATIONS	Under development/Testing Phase.
GENERATE_NOISE	Completed/Embedded in KALMAN_FILTER_TEST.
SIMULATE_TARGET	Completed/Embedded in KALMAN_FILTER_TEST.
KALMAN_FILTER_CKF	Completed/In Test.
KALMAN_FILTER_SCRF	Under development.
KALMAN_FILTER_CSRF	In Design Phase.
KALMAN_FILTER_SRIF	In Planning Phase.

VI. CONCLUSION

The wide use of Kalman filters for processing data in embedded computers and the variety of Kalman filter implementations available have made it desirable to place several such filters in a library of reusable Ada components. A study of the published literature and practical experience have led to the selection of the conventional Kalman filter (CKF), square root covariance filter (SRCF), Chandrasekhar square root filter (CSRF), and square root information filter (SRIF). In the development and testing of software systems, it is necessary to be able to test the performance of a particular filter under simulated conditions. Therefore, components which support system simulation have also been designed and tested for inclusion in the Ada library.

Procedures for interactive user input to, and control of, system simulations with Kalman filters have also been provided in the library. In addition to the mathematical functions that would be included in any Ada library, Ada packages for statistical functions, advanced matrix operations, and graphic displays have been produced.

Further work is planned to evaluate the run-time characteristics of this Ada software in embedded computer applications. It is anticipated that the documentation for these generic Ada Kalman filter packages will provide the user with measured run-time, memory-use, and accuracy data for particular embedded computer environments. The documentation will also describe Kalman filter performance in producing state estimates for selected system configurations.

The life cycle productivity of system and software engineering efforts will be considerably enhanced by the availability of Kalman filter packages and supporting Ada components in a library designed for convenient reusability.

ACKNOWLEDGEMENTS

The authors wish to express their appreciation to Theresa M. Bennett, Craig E. Kirkland, and Dr. Leung Ping Luk for their suggestions, comments, and assistance in completing the work on which this paper is based; to M. Louise Mills for editorial assistance; and to Edward C. Soistman, Jr., and Victor J. Sorondo for reviewing the paper, making helpful suggestions for its improvement, and providing support and encouragement during the development period.

REFERENCES

[1] Ada Joint Program Office, ADA PROGRAMMING LANGUAGE, Department of Defense, Military Standard ANSI/MIL-STD-1815A, Jan. 22, 1983.

[2] G. Booch, SOFTWARE COMPONENTS WITH ADA: STRUCTURES, TOOLS, AND SUB-SYSTEMS. Menlo Park, CA: Benjamin/Cummings, 1987.

[3] G. Booch, SOFTWARE ENGINEERING WITH ADA (Second Edition), (Series in Ada and Software Engineering). Menlo Park, CA: Benjamin/Cummings, 1987.

[4] G. Booch, "Object-oriented development", IEEE Trans. Soft. Eng., vol SE-12, pp. 211-221, Feb. 1986.

[5] J. R. Johnson and D. E. Johnson, LINEAR SYSTEMS ANALYSIS. New York: Ronald Press, 1975.

[6] R. E. Kalman, "Mathematical theory of optimization", in THE NEW ENCYCLOPAEDIA BRITANNICA: MACROPAEDIA, vol. 13, pp. 621-638, 1974.

[7] S. S. L. Chang, SYNTHESIS OF OPTIMUM CONTROL SYSTEMS (Series in Control Systems Engineering). New York: McGraw-Hill, 1961.

[8] A. E. Bryson and Y. C. Ho, APPLIED OPTIMAL CONTROL. Waltham, MASS: Blaisdell, 1969.

[9] A. Gelb, ed., APPLIED OPTIMAL ESTIMATION. Cambridge, MASS: M. I. T. Press, 1974.

[10] B. D. O. Anderson and J. B. Moore, OPTIMAL FILTERING, (Information and System Sciences Series). Englewood Clifs, NJ: Prentice-Hall, 1979

[11] R. E. Kalman, "A new approach to linear filtering and prediction problems", Trans. ASME (J. Basic Eng.), vol.82D, pp.34-45, Mar. 1960.

[12] I. B. Rhodes, "A tutorial introduction to estimation and filtering", IEEE Trans. Auto. Control, vol. AC-16, pp. 688-706, Dec. 1971.

[13] G. J. Bierman, "A comparison of discrete linear filtering algorithms", IEEE Trans.Aero. and Electronic Sys., pp. 28-37, Jan. 1973.

[14] H. W. Sorenson, "Guest editorial on applications of Kalman filtering", IEEE Trans. Auto. Control (Special issue on Applications of Kalman filtering), vol. AC-28, pp.254-255, Mar. 1983.

[15] C. E. Hutchinson and J. H. Fagan, "Kalman filter design considerations for space-stable inertial navigation systems", IEEE Trans. Aero. and Electronic Sys., vol. AES-9, pp. 306-319, Mar. 1973.

[16] J. S. Thorp, "Optimal tracking of maneuvering targets", IEEE Trans. Aero. and Electronic Sys., vol. AES-9, pp. 512-519, Jul. 1973.

[17] R. F. Berg, "Estimation and prediction for maneuvering target trajectories", IEEE Trans. Auto. Control, vol. AC-28, pp. 294-304, Mar. 1983.

[18] D. Williamson, "Finite wordlength design of digital Kalman filters for state estimation", IEEE Trans. Auto. Control, vol. AC-30, pp. 930-939, Oct. 1985.

[19 G. J. Bierman, FACTORIZATION METHODS FOR DISCRETE SEQUENTIAL ESTIMATION. New York: Academic, 1977.

[20] S. M. Bozic, DIGITAL AND KALMAN FILTERING. London: Edward Arnold, 1979.

[21] V. A. Downes and S. J. Goldsack, PROGRAMMING EMBEDDED SYSTEMS WITH ADA. Englewood Cliffs, NJ: Prentice-Hall, 1982.

[22] W. J. Cody, Jr., and W. Waite, SOFTWARE MANUAL FOR THE ELEMENTARY FUNCTIONS (Series in Computational Mathematics). Englewood Cliffs, NJ: Prentice-Hall, 1980.

[23] R. A. Frazer, W. J. Duncan, and A. R. Collar, ELEMENTARY MATRICES AND SOME APPLICATIONS TO DYNAMICS AND DIFFERENTIAL EQUATIONS. New York: Macmillan, 1946.

[24] R. Bellman, INTRODUCTION TO MATRIX ANALYSIS (Series in Matrix Theory). New York: McGraw-Hill, 1960.

[25] F. Ayres, Jr., THEORY AND PROBLEMS OF MATRICES (Schaum's Outline Series in Mathematics). New York: McGraw-Hill, 1962.

[26] A. S. Householder, THE THEORY OF MATRICES IN NUMERICAL ANALYSIS. Waltham, Mass: Blaisdell, 1964.

[27] K. Nomizu, FUNDAMENTALS OF LINEAR ALGEBRA. New York: McGraw-Hill, 1966.

[28] S. S. Wilks, MATHEMATICAL STATISTICS (Publications in Mathematical Statistics). New York: Wiley, 1962.

[29] M. R. Spiegel, THEORY AND PROBLEMS OF PROBABILITY AND STATISTICS (Schaum's Outline Series in Mathematics). New York: McGraw-Hill, 1975.

[30] J. E. Freund and R. E. Walpole, MATHEMATICAL STATISTICS (Third Edition). Englewood Cliffs, NJ: Prentice-Hall, 1980.

[31] B. Burkhardt and M. Lee, "Drawing Ada structure charts", Ada Letters, Vol. VI, No. 3, pp. 71-80, May-June 1986.

[32] R. St. Dennis, P. Stachour, E. Frankowski, E. Onuegbe, "Measurable characteristics of reusable Ada software", Ada Letters, Vol. VI, No. 2, pp. 41-50, March-April 1986.

[33] N. Wiener,THE EXTRAPOLATION, INTERPOLATION, AND SMOOTHING OF STATIONARY TIME SERIES. New York: Wiley, 1949.

[34] L. A. Zadeh and J. R. Ragazinni, "An extension of Wiener's theory of prediction", Jour. Applied Phys., vol 21, pp. 645-655, 1950.

[35] H. W. Bode and C. E. Shannon, "A simplified derivation of linear, least-squares smoothing and prediction theory", Proc. IRE, vol. 38, pp. 417-425, 1950.

[36] C. T. Leondes, editor, THEORY AND APPLICATIONS OF KALMAN FILTERING, NATO Advisory Group for Aerospace Research and Development, AGARDograph No. 139, 537 pages, Feb. 1970.

[37] C. T. Leondes, editor, ADVANCES IN THE TECHNIQUES AND TECHNOLOGY OF THE APPLICATION OF NONLINEAR FILTERS AND KALMAN FILTERS, NATO Advisory Group for Aerospace Research and Development, AGARDograph No.256, 538 pages, Mar. 1982.

[38] B. Gold and C. M. Rader, DIGITAL PROCESSING OF SIGNALS. New York: McGraw-Hill, 1969.

[39] A. V. Oppenheim and R. W. Schafer, DIGITAL SIGNAL PROCESSING. Englewood Cliffs, NJ: Prentice-Hall, 1975.

[40] L. R. Rabiner and B. Gold, THEORY AND APPLICATION OF DIGITAL SIGNAL PROCESSING. Englewood Cliffs, NJ: Prentice-Hall, 1975.

[41] A. Peled and B. Liu, DIGITAL SIGNAL PROCESSING: THEORY, DESIGN, AND IMPLEMENTATION. New York: Wiley, 1976.

[42] S. A. Tretter, INTRODUCTION TO DISCRETE-TIME SIGNAL PROCESSING. New York: Wiley, 1976.

[43] R. J. A. Buhr, SYSTEM DESIGN WITH ADA. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[44] J. G. P. Barnes, PROGRAMMING IN ADA (Second Edition), (International Computer Science Series). Reading, MASS: Addison-Wesley, 1984.

[45] D. G. McNicholl, C. Palmer, and others, "Kalman filter parts rationale", in COMMON ADA MISSILE PACKAGES (CAMP), Air Force Armament Laboratory Report AFATL-TR-85-93, vol. III, sec. V, pp. 13-26, May 1986.

[46] M. Verhaegen and P. Van Dooren, "Numerical aspects of different Kalman filter implementations", IEEE Trans. Auto. Control, vol. AC-31, pp. 907-917, Oct. 1986.

[47] J. F. Bellantoni and K. W. Dodge, "A square root formulation of the Kalman-Schmidt filter", AIAA Jour., vol. 5, pp.1309-1314, Jul. 1967.

[48] P. G. Kaminski, A. E. Bryson, Jr., and S. F. Schmidt, "Discrete square root filtering: a survey of current techniques", IEEE Trans. Auto. Control, vol. AC-16, pp. 727-736, Dec. 1971.

[49] R. H. Battin, ASTRONAUTICAL GUIDANCE. New York: McGraw-Hill, pp. 388-389, 1964.

[50] J. E. Potter, "A matrix equation arising in statistical filter theory", NASA Contract Rep., NASA CR-270, Aug. 1965.

[51] M. G. Salvadori, NUMERICAL METHODS IN ENGINEERING, (Civil Engineering and Engineering Mechanics Series), Englewood Cliffs, NJ: Prentice-Hall, 1952.

[52] D. C. Fraser, "A new technique for the optimal smoothing of data", M. I. T. Instrumentation Lab. Report T-474, Jan. 1967.

[53] T. Kailath, "Some new algorithms for recursive estimation in constant linear systems", IEEE Trans. Inform. Theory, vol. IT-19, pp. 750-760, Nov. 1973.

[54] E. L. Ince, ORDINARY DIFFERENTIAL EQUATIONS, New York: Dover, 1944.

[55] A. Andrews, "A square root formulation of the Kalman covariance equations", AIAA Jour., vol. 6, pp. 1165-1166, June 1968.

[56] R. G. Stanton, NUMERICAL METHODS FOR SCIENCE AND ENGINEERING (Applied Mathematics Series). Englewood Cliffs, NJ: Prentice-Hall, 1961.

[57] B. Carnahan, H. A. Luther, and J. O. Wilkes, APPLIED NUMERICAL METHODS. New York: Wiley, 1969.

[58] E. K. Blum, NUMERICAL ANALYSIS AND COMPUTATION THEORY AND PRACTICE (Series in Mathematics). Reading, MASS: Addison-Wesley, 1972.

[59] R. W. Hamming, NUMERICAL METHODS FOR SCIENTISTS AND ENGINEERS (Second Edition), (International Series in Pure and Applied Mathematics). New York: McGraw-Hill, 1973.

[60] R. W. Hornbeck, NUMERICAL METHODS. Englewood Cliffs, NJ: Prentice-Hall, 1975.

[61] R. L. Burden, J. D. Faires, and A. C. Reynolds, NUMERICAL ANALYSIS (Second Edition). Boston: Prindle, Weber & Schmidt, 1981.

[62] M. Morf, G. S. Sidhu, and T. Kailath, "Some new algorithms for constant linear, discrete-time systems", IEEE Trans. Auto. Control vol. AC-19, pp. 315-323, Aug. 1974.

[63] M. Morf and T. Kailath, "Square root algorithms for least-squares estimation", IEEE Trans. Auto. Control, vol. AC-20, pp. 487-497, Aug. 1975.

[64] C. G. Boncelet, Jr., and B. W. Dickinson, "An extension to the SRIF Kalman filter", IEEE Trans. Auto. Control, vol. AC-32, pp. 176-179, Feb. 1987.

[65] J. L. Lawson and G. E. Uhlenbeck, THRESHOLD SIGNALS. New York: Dover, 1950.

[66] S. F. Schmidt, "Computational techniques in Kalman filtering", in THEORY AND APPLICATIONS OF KALMAN FILTERING, C.T. Leondes, ed., NATO Advisory Group for Aerospace Research and Development, AGARDograph 139, pp. 65-86, Feb. 1970.

[67] R. A. Singer, "Estimating optimal tracking filter performance for manned maneuvering targets", IEEE Trans. Aero. and Electronic Sys., vol AES-6, pp. 473-483, Jul. 1970.

[68] R. J. Fitzgerald, "Divergence of the Kalman filter", IEEE Trans. Auto. Control, vol. AC-16, pp. 736-747, Dec. 1971.

[69] J. M. Mendel, "Computational requirements for a discrete Kalman filter", IEEE Trans. Auto. Control, vol. AC-16, pp. 748-758, Dec. 1971.

[70] N. A. Carlson, "Fast triangular formulation of the square root filter", AIAA Jour., vol. 11, pp. 1259-1265, Sep. 1973.

BIOGRAPHIES

Sherrie A. Branyon was born in Pryor, Oklahoma, on June 3, 1938. She received the B.S. degree in Mathematics from Rollins College, Winter Park, Florida, in 1967. She taught high school mathematics in Orlando, Florida, from 1967 to 1978. Under a National Science Foundation grant, she earned an M.S. degree in Mathematics from the University of South Carolina in 1973. Her experience in the aerospace industry has been at the Martin Marietta Electronics and Missiles Group, Orlando, Florida, in software design for real-time, embedded systems applications.

She developed an Ada training course for Martin Marietta in Orlando, and has been teaching Ada classes to software engineers since 1984. She also served on the STARS workshop committee to produce an Ada Reusability Guidebook. She is currently supervising the Ada Technology Support Center for Martin Marietta in Orlando.

George K. Hess, Jr., was born in Orange, New Jersey, on August 27, 1922. He earned B.S. degrees in Engineering Mechanics and Mathematics in 1945, the M.S. degree in Engineering Mechanics in 1947, and the Ph.D degree in Engineering Mechanics in 1954, from the University of Michigan. During World War II, he was commissioned as Ensign, USNR. His work experience includes a variety of subjects including, hydrogen bomb tests, ship vibrations, cryogenic system design, air-air missile guidance, nuclear rocket engine controls and tests, ballistic missile range instrumentation, interdiction sensor development, land combat system design, magnetic fusion energy research, analog and digital computer programming, and program management.

Dr. Hess is a member of ASME, AIAA, IEEE, ARRL, Sigma Xi, and Tau Beta Pi. He received the Miller Award in Mathematics at the University of Michigan. He has earned several Outstanding Performance awards from the USAF and the US Army. He received the Exceptional Civilian Service award from the USAF for work on missile range tracking and telemetry ships.

Henry B. Knowles, Jr. was born in Baltimore, Maryland, on January 11, 1962. He earned the B.S. degree in Computer Science, Mathematics, and Statistics in 1984, and the M.S. degree in Computer Science in 1987, from the University of Central Florida. As a Graduate Research Assistant, he participated in an Ada Risk Assessment project sponsored by the Naval Training Systems Center in Orlando. He is currently working in the Ada Technology Support Center at Martin Marietta Electronics and Missiles Group in Orlando. His primary responsibilities include the development of reusable Ada components and the support of projects involving the Ada programming language.

A KNOWLEDGE-STRUCTURE OF A REUSING SOFTWARE COMPONENT IN LIL

By Dar-Biau Liu

Department of Computer Science and Engineering
California State University, Long Beach
Long Beach, California 90840

ABSTRACT

In this paper, we will address the problem of accessing effectively the software components in LIL's Library by combining both formal and informal specifications methods to support reusability in Ada programming environment. The ANNA-like modified LIL specification language will be used. A knowledge-structure for a software component in LIL's Library, as introduced in [2], will be imposed. It allows a description of the domain of variables, and describes the resource characteristics of a component. It also provides the capabilities and facilities for users to determine, and ways to search for the appropriate resuable software component in LIL's Library.

I. INTRODUCTION

One of the most important goals in any programming environment is to make programming significantly easier, more reliable, and cost effective. In [5], [6], we examined the problem of making verifiable specifications in an Ada software development environment, then illustrated the program development methodology of analogous abstraction and finally, illustrated how the verifiable specifications can aid in making analogous abstractions in the Ada programming environment. LIL, introduced by Goguen in [3], was used to do all these works. But how can software components in LIL be accessed effectively by the programmer?

In this paper, we will address this problem by combining both formal and informal specifications methods in LIL's Library to support reusability in Ada programming environment. The ANNA-like modified LIL specification language will be used. A knowledge-structure for a software component in LIL's Library, as introduced in [2], will be imposed. It allows a description of the domain of variables, and describes the resource characteristics of a component. It also provides the capabilities and facilities for users to determine, and ways to search for the appropriate resuable software component in LIL's Library.

By combining the formal and informal specification methods, we have a powerful tool to support the reusability in Ada programming environment. While informal specification method itself can only provide the functional description of a package and a mean to enable a user to allocate a particular software component of interest in the LIL's Library; to include the ANNA-like modified LIL formal specifications, we can analyze mathematically to determine the correctness of the specification and for possible equivalence between different sets of specifications, we can also check for their consistency and completeness. But most important of all, it can be automatically processed by a computer.

II. LIL SPECIFICATION LANGUAGE

As discussed in [3], [5], [6], LIL uses THEORIES containing semantics to formalize data abstractions (sets, variables, functions, etc...), THEOREIES may use other THEORIES and etc. to extend their properties and to bind them together. We will visualize LIL's THEORIES as the software development system primitives; the GENERIC THEORIES will be emboddied in making parameters, i.e., it will mean adding structures and semantics to existing THEORIES. GENERIC PACKAGES are used for ADT (abstract data types) and functions (or procedures). Further, LIL utilizes the "make" command using the concept "view" of THEORIES and PACKAGES to make an instantiation of an existing software component in LIL's Lbirary.

III. AN EXAMPLE

Figures 1 and 2 in the appendix is an example of an generic unit with ANNA-like

*This research is partially funded by the California State University, Long Beach, California.

modified LIL formal specifications and informal specifications included. It is for demonstration only, it is by no means complete.

The generic unit in the appendix can be described as an abstract data type (ADT) in a LIL's Library. An ADT with informal specifications consists of a list of aliases, a natural language description, a list of keywords, a domain specification, and zero or more operation specifications. Syntactically, aliases, descriptions and keywords are represented by the special comment symbol "--", which are optional, but if they do exist, they must appear immediately after the package header. On the other hand, if ANNA-like formal specification is included, it may appear anywhere in a package wherever valid Ada texts are allowed.

Aliases are synonym names for the ADT name. In Figure 1, the ADT name is "stack", and its aliases are LIFO-LIST, LIFO, and LIST (or its analogous equivalences).

The natural language description describes an ADT; it has no format or content constraints, but it should be written to be as helpful as possible for a programmer to identify the ADT.

The keywords allow context words or phrases be specified (e.g. LIFO, LAST-IN-FIRST-OUT).

The domain specification is a regular expression whose elements are ADTs, the primitive elements defined by THEORIES, or ADT instantiation parameters of the ADTs that had been defined, or those parameters specified by the GENERIC THEORIES in the LIL's Library. Ada does not allow a domain specification to be syntactically represented. the domain implementation for the package "stack" can be stored separately from the ADT specification (Ada allows this). The domain specification thus defines a value set for an ADT. It is a mapping from the value set of an ADT to a set of representation defined in terms of previously existing ADTs (GENERIC THEORIES OR GENERIC PACKAGES) in the LIL's Library, some of which may be given as ADT instantiation parameters. Figure 2 is an example of the domain implementation for the package "stack". It is specified as an array of ELTs (a THEORY in the LIL's Library). Note that the very first line contains the characters "#1", which is to differentiate the different domain implementations for a given ADT.

Each operation in the operation specification is specified by giving its name, arguments, results, and exceptions if any. following each operation specification is an optional list of operation aliases, a description, and a list of keywords identical to the sections of the package header.

As in example (Figure 1), in the operation section, the special commented line "--: Function length Return natural;" and its associated lines with the symbol "--¦" represent ANNA-like formal specifications. The function "length" in the package "stack" is an example of the virtual concept in ANNA. It may be viewed as a concept the programmer used in formulating and designing the package. It is a "virtual function," it is not visible in the actual Ada source (text). It is used to specify the input and output conditions on procedures "PUSH" and "POP". The ANNA-like annotation "--¦" for the function "length" could also be declared in the body of the package "stack" for run-time checking of correctness of calls to "PUSH" and "POP".

From this example, it is obvious that this ANNA-like annotations support program debugging, verification of specifications and checking program performance against axioms and against its specifications.

IV. ADT RELATIONSHIPS

The followings are only a few of the relationships among the ADTs, adopted from [2], in a LIL's Library (serious reader should refer to [2] for more ADT relationships):
1. Depend-On(a,b,i): ADT "a" depends on ADT "b" in the implementation "i" if and only if "i" is an implementation of "a" and "i" references to "b". For example, (stack, LIFO, #1) holds. With Depend-On relation, it is possible to recursively list all ADTs needed for the implementation of a particular ADT. It is thus possible to build automatically a complete, compilable software component for an ADT in a LIL's Library.
2. Close(a, b, n): ADTs "a" and "b" are n-close if they differ by at most "n" operations. This makes it possible to move from ADT "a" to n-close ADT "b". This relationship can be used to compare differences in domains and operations of ADTs in a LIL's Library.

V. HOW TO USE LIL'S LIBRARY

Based on the relationships derived and the knowledge structure for each software component (e.g. ADTs) in the LIL's Library, a programmer can find a particular component of interest by name, or by automatic use of aliases. General description provided by a programmer can be compared with the natural language descriptions associated with each component. The keywords list can be used for searching an ADT, this yields a faster search in comparing to the use of natural language descriptions. A programmer can also locate the desired component by providing partial domain definitions. The system would then search for any component (e.g. ADT) with a domain specification containng some or all of the giving domain specifications.

VI. SUMMARY STEPS

Summary steps for finding a component (ADT) of interest from LIL's Library:

1. Use the keywords search, natural language description comparison, or search by name through aliases, a programmer can find a list of components (ADTs) of possible candidates.
2. Determine the type of operations that is needed for particular implementation.
3. Assuming that such functions as "find", "searcn", "browse" or "n-close" exist in the system, a programmer can use the informations obtained from steps 1 & 2 to find the software component of interest in the LIL's Library.
4. Once the desired component is found, an ANNA-like LIL preprocessor will be used to translate this component into Ada texts that checks whether the assertion is satisfied by a program stated. This checking can be executed together with the underlying Ada program, thus performing run-time checking.
5. The component (ADT) found in step 4 can be compiled via a standard Ada compiler. The resulting software system, when executed, enables an Ada program to be tested for consistency with their formal specifications, thus support reusability.

CONCLUSION

We believe that by using the program development methodology of analogous abstractions, and then storing these abstractions in the LIL's Library by combining both formal and informal specification methods, using the ADT relationships and the access tools as we summarized in sections IV, V, VI above and as in [2], we have developed a powerful tool for reusability in Ada programming environment.

APPENDIX

EXAMPLE:-

```
  -- Any needed "with" statements go here.
Generic
  type ELT is private;
  Size: IN Natural;
Package.Stack is
  -- Aliases: LIFO-LIST, LIST, LIFO.
  -- Description: This stack is a last-in-first-
     out stack of at most "size" ELTs.
  -- Keywords: LIFO, LAST-IN-FIRST-OUT, LIST
  -- Domain:
  -- ELT (0 - size).
type definition is privtae;
  -- Operations:
  Max: natural: = 100
  Min: natural: = 0,
  --| Where In size < Max;
Function Create Return Stack. definition;
  -- Aliases; New, Initialize.
  -- Description: Create the stack with 0 ELT in
     it.
  --: Function length Return natural;
Procedure PUSH (New_ELT: IN ELT);
  --| Where In stack.length < Max,
  --| Out(stack.length) = In(stack.length) + 1;
Procedure POP (Old_ELT. OUT ELT);
  --| Where In stack.length < Min,
  --| OUT(stack.length) = In(stack.length) - 1;
Function Empty Return ELT;
Function Top Return ELT;
Exceptions
  Stack-underflow;
  Stack-Empty;
S : Stack.definition;
I : ELT;
  --' Axioms:
  --:     (POP(PUSH(S,I) = S)
  --|     (TOP(PUSH(S,I) = I)
  --:     (EMPTY(CREATE) = True)
  --|     (POP(EMPTY) = Stack-underflow)
  --'     (TOP(EMPTY) = Stack-Empty)
End Stack,
  -- Any neede "with" statements go here.
Package body Stack is
  Function Create Return Stack.definition is..;
  Procedure PUSH(New_ELT: IN ELT) is ....
  Procedure POP(OLD_ELT: OUT ELT) is...;
  Function EMPTY Return Boolean is...;
End Stack;
```

Figure 1

```
  -- Any needed "with" statements go here.
  -- Domain Implementation #1 of Stack.
Private
  type List is array(1...size) of ELT;
  type LIFO-LIST.definition is record
        LIFO: LIST;
        Front : Integer Range 1...size;
        Back : Integer Range 1....size;
  End record;
End Stack;
```

Figure 2

REFERENCES

1. Booch, G.: Software Engineering With Ada, Menlo Park, Benjaman/Cummings, 1983.

2. Embley, D. W. & Woodfield, S. N.: A Knowledge-Structure for Reusing Abstract Data Types in Ada Software Production, Proceedings of the Join+ Ada Conference, 5th National Conference on Ada Technology and Washington Ada Symposium, March 1987.

3. Goguen, J. A. : Reusing and Interconnecting software components, IEEE Computer, Vol. 19, No. 2, pp. 16-28, Feb. 1986.

4. Litvichouk, S. D. & Matsumoto, A. S.: Design of Ada System Yielding Reusable Components: An Approach Using Algebraic Specification, IEEE Transaction of Software Engineering, Vol. SE-10, No. 5, Sept. 1984.

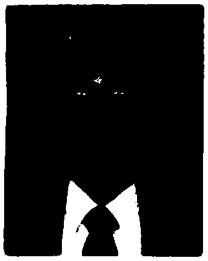
5. Liu, D. B. & Harrison, G.: Generic Implementation via Analogous in the Ada Programming Language, ACM Ada Letters,

Vol. VI, No. 4, pp. 34-43.

6. Liu, D. B. and Harrison, G.: LIL as a Resuable tool for the Ada Programming Language, Proceedings of International Computer Symposium, Dec., 1986.

7. Luckham, D. C. & Von Henke, F. W.: An Overview of ANNA, A Specification Language for Ada, IEEE Software, Vol. 2.2, pp. 9-24, March 1985.

8. Owen, G. S., Gagliano, R. & Honkanen, P.: Functional Specifications of Reusable MIS Software in Ada, Proceedings of the Joint Ada Conference, 5th National Conference on Ada Technology and Washington Ada Symposium, March 1987.



Dar-Biau Liu is a Professor at California State University, Long Beach where he has been a faculty member since August 1986. He teaches graduate and undergraduate classes in Software Engineering, Distributed Computer System, Computing theory and Programming Methods. His research interests include Software Reusability, Object-Oriented Design, and Dynamic task scheduling in Distributed Computer System. Before coming to California State University, Long Beach he was a faculty member at Old Dominion University, Norfolk, Va. Previously, he was a Staff Engineer at IBM Corp. and was a project manager at ITT Corp. He received a Ph.D. in Applied Mathematics and Computer Science at the University of Wisconsin-Madison in 1972. Previously he had received a M.A. in Mathematics from Wayne State University, and a B.S. in Mathematics from National Taiwan Normal University. His current address is: Department of Computer Science and Engineering, California State Univertsity, Long Beach, Ca 90840.

REUSABLE ADA MODULES

FOR

ARTIFICIAL INTELLIGENCE APPLICATIONS

by

Verlynda S. Dobbs
Wright State University

ABSTRACT

The development of reusable Ada modules as one component of an Ada artificial intelligence toolkit is described. Modules for bidirectional heuristic search are implemented and used successfully in two domains: sliding block puzzle and flight path generator. The reusability provided by the modules is at both the code and specification levels.

1.0 INTRODUCTION

The use of Ada for artificial intelligence (AI) applications could be greatly facilitated by the existence of an Ada AI toolkit. This toolkit would contain components such as libraries of reusable modules, code generators and expert system shells. This work concentrates on one toolkit component, reusable Ada modules, with experiences developing heuristic search modules.

2.0 REUSABILITY EFFORTS

Reusability can be simply defined as the ability of an element to be used again. In this spirit, the terms "software reuse" and "software reusability" have been applied to a spectrum of techniques which include classical portability, shared code, repeated use of algorithms and the incorporation of building blocks. [LEN87] Attempts to analyze reusability have resulted in numerous categorizations of the elements. These include, but are not limited to, the abstraction level of the element (requirements, design, data, code) [BOO83, FRE83]; the reusability level of the element (weak, strong, effective) [GAR87]; and the level of customization required for use of the element (none, manual modification, templates for modification, generic parameterization) [STA83]. Currently, much attention is focusing on methods for cataloging and describing the contents of libraries of reusable software modules. [HEN87, FIS87]

Although reusability is a characteristic often associated with Ada, there has been minimal practical experience with reusing Ada modules and no generally accepted methodologies for reuse. Several Ada projects are currently under development that would facilitate AI efforts if the modules could be provided in a reusable library. These include tools for pattern-directed string processing [REE85], a method for implementing semantic networks [SCH86], and the modules described in this paper for heuristic search.

3.0 REUSABLE MODULES FOR HEURISTIC SEARCH

Elements for reuse must be chosen carefully. The elements must embody common concepts and abstractions from domains which have reached a degree of maturity. Heuristic search algorithms, such as A*, [HAR68] have been widely published in artificial intelligence textbooks and are well understood. These algorithms can be used in many application areas where heuristic search is desirable. The algorithm used in these modules is for bidirectional heuristic search (DNODE) [POL84], a search from a known start state to a known goal state.

3.1 Heuristic Search

Searching has been traditionally performed blindly, either breadth first or depth first. These blind searches continue to generate all successors of nodes (expand) in some predetermined order until a goal node is found. An exponential explosion occurs in both the time and the space required to find a solution.

The size of the search tree can be reduced by using more informed search techniques. Heuristics (rules of thumb) can be used to estimate a new node's distance from a goal. The next node chosen for expansion will then be the node believed to be the closest to a goal. Consequently, time and space will not be wasted on nodes that show very little promise.

To help avoid an exponential explosion, a heuristic function h, whose two arguments are state space nodes, is provided. h returns an estimate of the path cost between its arguments and is used by the search algorithm to determine the best node to expand next.

Heuristic search algorithms take one of two basic approaches. Unidirectional search algorithms, which build one search tree of nodes rooted at the start, expand nodes from the start until a goal node is found. Bidirectional search algorithms maintain two search trees of nodes rooted at the start and the goal. The nodes on these trees are those so far generated by the algorithm, and the path from a root to a leaf represents the least cost path the algorithm has yet found connecting root and leaf. The leaves, called open nodes, are candidates for expansion. The algorithm attempts to expand leaves in such a pattern that the two trees "meet" as soon as possible, at which point the problem is solved and a route from start to goal may be reported.

In order to determine which open node n to expand next, there is an <u>evaluation function</u> of the form

$$f(n) = (1 - w) * g(n) + w * h_o(n), \quad 0 <= w <= 1.$$

Here g(n) is the cost of the search tree path from node n to its corresponding root, h_o is the heuristic component of f (derived in some fashion from the function h mentioned above), and w is a weight fixed to some value deemed to be effective. Small values of f(n) indicate that n is more likely to be on a lost cost solution path than do large values. During each cycle of the search algorithm, an open node with minimal f-value is selected from the appropriate search tree for expansion. Its successors are generated. If one of the successors is on the opposite search tree, the algorithm halts, otherwise f is evaluated for each of the successors and they are appropriately placed on a search tree. The algorithm recycles.

3.2 Module Development

Development of reusable modules for heuristic search concentrated on keeping all the <u>domain dependent</u> information separate from the search strategy. The result is a set of packages that encapsulate the search graph, the search strategy, the heuristics, and the domain. All the program units are shown in Figure 1. An arrow between two units indicates that one unit depends upon elements (data, procedures, or objects and operations) contained in the other unit. The unit enclosed in a rectangle is used by all the other units.

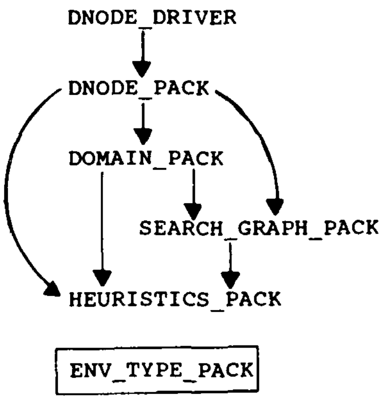


Figure 1. Relationship of Units

Three of the program units are <u>domain independent</u>. These units are the main program (DNODE_DRIVER) and the packages encapsulating the search graph (SEARCH_GRAPH_PACK) and the search strategy (DNODE_PACK). The main program unit requests input from the user of the number of problems to be solved and whether to trace the program. The search graph package is an abstract data type that contains object definitions for a search graph and the operations available on those objects (standard list operations plus other operations specific to a search graph). The search strategy package encapsulates a collection of procedures for bidirectional heuristic search. Only one procedure is visible from outside the package, the heuristic search procedure (DNODE). All of these domain independent program units can be used in other domains without modification.

Parts of the remaining three units must be provided for each domain. These units describe the problem state (ENV_TYPE_PACK), the domain (DOMAIN_PACK), and the heuristics (HEURISTICS_PACK). The problem state unit must contain the type definition for a state of the problem.

The domain unit contains all procedures required for the domain except the heuristics, which are contained in the heuristics unit. Since the specifications are given for both the domain and the heuristics, only the bodies must be completed. The specifications of HEURISTICS_PACK and DOMAIN_PACK are shown in the Appendices A and B, respectively.

3.3 Module Use

The modules can be used by completing the following steps.

3.3.1 Step 1: Identify domain

The domain in which the heuristic search is to be used must be identified.

3.3.2 Step 2: Complete ENV_TYPE_PACK

To complete ENV_TYPE_PACK, two data structures must be chosen. The first represents a state of the domain (of type STATETYPE), and the second contains variables relevant to the domain (in a record of type DOMAIN_TYPE). A constant (MAX) is set to the maximum number of new states that can be generated from any state. Any domain specific constants are also declared. After ENV_TYPE_PACK has been compiled, the following units must then be recompiled in the order shown.

HEURISTICS_PACK specification
SEARCH_GRAPH_PACK
DOMAIN_PACK specification
DNODE_PACK
DNODE_DRIVER

(The bodies of HEURISTICS_PACK and DOMAIN_PACK must be compiled after the code has been completed.)

3.3.3 Step 3: Complete HEURISTICS_PACK

Heuristics must be identified that have been found to be effective in the domain. All procedures used to implement the heuristics are included in HEURISTICS_PACK. Three procedures are visible from outside HEURISTICS_PACK: SET_G_VAL, SET_H_VAL, and INIT_H. SET_G_VAL and SET_H_VAL each have three parameters: a selector (HSEL) used to select the desired heuristic and two parameters that are both states of the problem (type STATETYPE). INIT_H initializes the heuristics.

3.3.4 Step 4: Complete DOMAIN_PACK

DOMAIN_PACK contains procedures to input and output a problem state, set a default state, open and close files, calculate statistics, output results of the search, and expand a node. These procedures must all be provided. An input file, opened in OPEN_FILES, contains the goal state followed by any number of start states. EXPAND generates all the successors of a problem state by applying all valid operations. This procedure, which is conceptually domain dependent, may need some search information. This will depend upon the domain, as shown in the applications in Section 4.

3.4 Reusability Characteristics

The reusability aspects of the units are summarized in Table 1 for the abstraction level and the level of customization required. The reusability levels remain to be evaluated by additional use of the packages for bidirectional heuristic searching in other domains.

Table 1. Reusability Characteristics

ELEMENT	ABSTRACTION LEVEL	CUSTOMIZATION
ENV_TYPE_PACK	Definitions	Manual*
HEURISTICS_PACK	Specification	Manual*
SEARCH_GRAPH_PACK	Code & Specification	None
DOMAIN_PACK	Specification	Manual*
DNODE_PACK	Code	None
DNODE_DRIVER	Code	None

*Templates for customization could be provided

4.0 APPLICATIONS

The modules were originally developed in Vax Ada on a Vax 11/780 running the VMS operating system. Sets of packages for two domains were used with the domain independent packages - sliding block puzzle and flight path generator. Development of these two sets of packages is described in the following sections by applying steps 2-4 of the procedure for using the modules (Section 3.3).

4.1 Sliding Block Puzzle

The specific sliding block puzzle used here is the 8-puzzle. The 8-puzzle consists of 8 numbered, movable tiles in a 3 by 3 matrix. One cell of the matrix is always empty, therefore an adjacent numbered tile can be moved into the empty cell. Two configurations are shown in Figure 2. The object of the search in the sliding block puzzle domain is to find a sequence of moves that will begin at the start state and end at the goal state.

06	01	[]		01	02	03
04	05	02		04	05	06
03	07	08		07	08	[]
	START				GOAL	

Figure 2. 8-puzzle Configurat ions

Steps 2-4 for the 8-puzzle are described in the following sections.

4.1.1 Step 2: Complete ENV_TYPE_PACK

The problem state for a sliding block problem can be represented as an array of characters. The only domain information needed is the number of tiles plus a constant for the maximum number of new states that can be generated from a problem state (4). ENV_TYPE_PACK for the 8-puzzle is shown in Figure 3.

```
PACKAGE ENV_TYPE_PACK IS

  TYPE STATETYPE IS ARRAY (1..8) OF
                    CHARACTER;
  TYPE DOMAIN_TYPE IS
       RECORD
            TILES : INTEGER;
       END RECORD;

  MAX : CONSTANT INTEGER := 4;

END ENV_TYPE_PACK;
```

Figure 3. ENV_TYPE_PACK for 8-puzzle

4.1.2 Step 3: Complete HEURISTICS_PACK

Several heuristics have been found to be effective in the sliding block puzzle domain. The Manhatten distance, based on the distance of a tile from its goal position, is used. INIT_H initializes a matrix of these distances. SET_H_VAL sums the distances of all the tiles in the current state from their positions in the deepest node on the opposite search tree. SET_G_VAL is the number of moves from the start state to the current state.

4.1.3 Step 4: Complete DOMAIN_PACK

Most of the procedures of DOMAIN_PACK are straight forward. The one interesting procedure is EXPAND. The four operations that can be applied to the current state are to move a tile up, down, left or right. If the resulting state is valid, the new state is added to the search graph. EXPAND for the 8-puzzle is shown in Appendix C.

4.1.4 Sliding Block Puzzle Summary

Use of the modules is straight forward for the sliding block puzzles. It is easy to keep the domain information and the search information separated.

4.2 Flight Path Generator

The problem space for a flight path generator (FPG) can be represented as a grid aligned along the x-y axis. The FPG is given a starting location (the start), a destination location (the goal), and a description of threats to be avoided (the costs associated with the edges of the graph). The search finds a flight path from the start to the destination based on avoiding the threats and producing a short path. An example is shown in Figure 4.

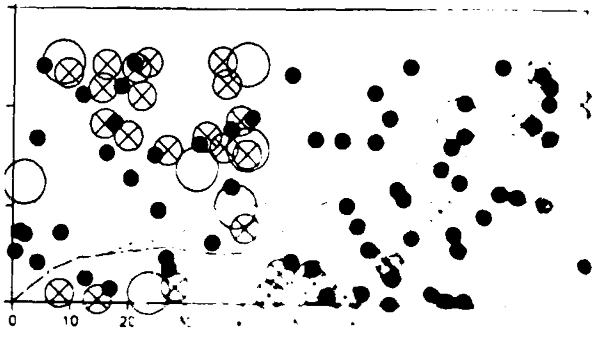
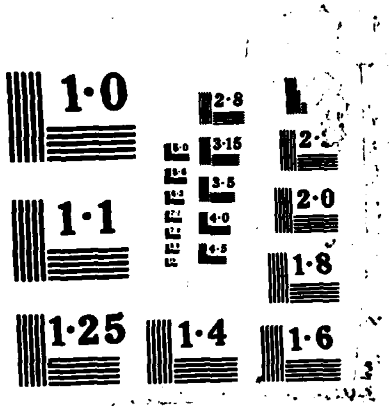


Figure 4 [illegible]

AD-A190 936 PROCEEDINGS OF THE NATIONAL CONFERENCE ON ADA (TRADE MARK) TECHNOLOGY (6TH) HELD ON 14-18 MARCH 1988 IN ARLINGTON VA(U) NORFOLK STATE UNIV VA MAR 88 5/7

UNCLASSIFIED F/G 12/5 NL


1·0
2·8
3·15
3·5
4·0
4·5
2·2
2·0
1·1
1·8
1·25
1·4
1·6

Steps 2-4 for the flight path generator are described in the following sections.

4.2.1 Step 2: Complete ENV_TYPE_PACK

The problem state for the flight path generator is a point on a grid. This point can be represented as a record with two components: the x-coordinate and the y-coordinate. Many constants and variables associated with the domain must be declared as shown in Figure 5. Several functions on the statetype are also provided.

```
PACKAGE ENV_TYPE_PACK IS

 TYPE STATETYPE IS
      RECORD
           X_COORD, Y_COORD : FLOAT;
      END RECORD;

 MAX : CONSTANT INTEGER := 15;
 MAX_THREAT_CATS : CONSTANT INTEGER
                                 := 5;
 MAX_THREATS : CONSTANT INTEGER := 150;
 MAX_LEGS : CONSTANT INTEGER := 15;
 PI : CONSTANT FLOAT := 3.141592654;
 DEG_RAD : CONSTANT FLOAT :=
                      0.01745329252;

 TYPE DOMAIN_TYPE IS
      RECORD
           X_LIMIT, Y_LIMIT : INTEGER;
           NR_THREATS, LEGS_PER_ARC :
                INTEGER;
           LEG_LENGTH, PATH_ARC,
                LEG_ARC, PATH_LENGTH,
                CONFLICT_COST : FLOAT;
           THREAT_SET : INTEGER;
      END RECORD;

 FUNCTION FIVE_SIGNIF (N : FLOAT)
      RETURN FLOAT;

 FUNCTION DISTANCE (POINT_A, POINT_B
      : STATETYPE) RETURN FLOAT;

 PROCEDURE INIT_DOM (LEG_LENGTH
      : FLOAT);

END ENV_TYPE_PACK;
```

Figure 5. ENV_TYPE_PACK for Flight Path Generator

4.2.2 Step 3: Complete HEURISTICS_PACK

A heuristic proposed by Lizza [LIZ84] was adapted for the flight path generator. The heuristic calculation for a state is obtained from the sum of the cost of threats intersected by a straight line drawn between two states. INIT_H initializes many parameters which effect the quality of the paths generated and contribute to the complete heuristics. These include the leg length, the path deflection angle, the maximum number of possible legs generated from each state within the angle of deflection, and others. SET_H_VAL sums the weighted cost of threats intersected by a straight line from the current state to the state of the deepest node on the opposite search tree. This sum is added to the distance between the two states. SET_G_VAL is the actual distance plus the cost of threats actually intersected by the path obtained from the start state to the current state.

4.2.3 Step 4: Complete DOMAIN_PACK

Expanding a state for the flight path generator is based on several parameters. A set number of legs are generated of a specified length within the angle of deflection. These new states are checked for validity before adding them to the search graph. EXPAND for the flight path generator is shown in Appendix D.

4.2.4 Flight Path Generator Summary

In the flight path generator domain, it was more difficult to separate all the domain independent and domain dependent information. Specifically, EXPAND needed information about the deepest nodes in the two search trees. Other difficulties were related to the continuous nature of the states of the flight path generator domain, in contrast to the discrete states of the sliding block puzzle.

5.0 CONCLUSIONS

The modules have been successfully used for bidirectional heuristic search in two domains. Many improvements remain to be made before the modules could be included in a library of reusable modules for public use. These include separating the domain dependent procedure EXPAND from the search strategy and providing module and parameter descriptions to facilitate the use of the modules.

ACKNOWLEDGEMENT

This research was sponsored by the Air Force Office of Scientific Research/AFSC, under Contract F49620-85-C-0013.

REFERENCES

BOO83 Booch, G., Software Engineering with Ada, Benjamin Cummings Publishing, Menlo Park, CA, 1983.

FIS87 Fischer, G., "Cognitive View of Reuse and Redesign," IEEE Software, July, 1987.

FRE83 Freeman, P., "Reusable Software Engineering: Concepts and Research Directions," Proceedings of the Workshop on Reusability in Programming, ITT, Stratford, CT, 1983.

GAR87 Gargaro, A. and T. Pappas, "Reusability Issues and Ada," IEEE Software, July, 1987.

HAR68 Hart,P., N. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," IEEE Transactions on Systems Science and Cybernetics, *SSC-4 (2), 1968.*

HEN87 Hendler, J.,Y. Wong, A. Vinciguerra and J. Mogilensky,"AIRS: An AI-based Ada Reuse Tool," The Third Annual Conference on Artificial Intelligence and Ada, October 14-15, 1987.

LEN87 Lenz, M., H. Schmid and P. Wolf, "Software Reuse Through Building Blocks," IEEE Software, July, 1987.

LIZ84 Lizza, C., "Generation of Flight Paths Using Heuristic Search," Masters Thesis, Wright State University, 1984.

POL84 Politowski, G. and I. Pohl, "D-node Retargeting in *Bidirectional* Heuristic Search," Proceedings of the National Conference on Artificial Intelligence, 1984.

REE85 Reeker, L., J. Kreuter and K. Wauchope, "Artificial Intelligence in Ada: Pattern-Directed Processing," AFHRL-TR-85-12, May, 1985.

SCH86 Scheidt, D., D. Preston and M. Armstrong, "Implementing Semantic Networks in Ada," 2nd Annual Conference on Artificial Intelligence and Ada, Fairfax, VA, November 12, 1986.

STA83 Standish, T., "Software Reuse," *Proceedings of the Work-shop on* Reusability in Programming, ITT, Stratford, CT, 1983.

Dr. Dobbs received her Ph.D. in computer science from The Ohio State University in 1985. Her research interests are in the areas of software engineering, artificial intelligence, and Ada for artificial intelligence. Dr. Dobbs is currently on the faculty of the Department of Computer Science and Engineering at Wright State University, Dayton, Ohio 45435.

APPENDIX A. HEURISTICS_PACK Specification

```
package heuristics_pack is

function set_h_val (hsel : in integer; pstate, qstate : in statetype)
        return float;

function set_g_val (hsel : in integer; pstate, qstate : in statetype)
        return float;

procedure init_h ( dom : in out domain_type);

end heuristics_pack;
```

APPENDIX B. DOMAIN_PACK Specification

```
package domain_pack is

procedure input_state (state : out statetype; p : in out ptr;
                       dom : in domain_type);

procedure output_state (dom : in domain_type; p : in ptr);

procedure set_default_state (dom : in domain_type; p : in ptr);

procedure expand (s : in out sys_rec_type; dn : in out dnode_type;
                  dom : in out domain_type);

procedure show_results (dom : in domain_type;
                        dn : in dnode_type; s : in sys_rec_type);

procedure open_files (bug : in boolean);

procedure close_files;

end domain_pack;
```

APPENDIX C. EXPAND for Sliding Block Puzzle

```
procedure expand (s : in out sys_rec_type; dn : in out dnode_type;
                  dom : in out domain_type) is
cur, p : ptr;
type move_direction is (up,down,left,right);
i, new_pos, start, bspace : integer;
state : statetype;
move : move_direction;
valid : boolean;
begin
    debug := s.debug;
    if debug then put_line ("entering expand");
        for i in 1..9 loop
            put (dn.current.state(i));
        end loop;
        new_line;
    end if;
    cur := dn.current;
    bspace := 1;
    while cur.state(bspace) /= blank loop
        bspace := bspace + 1;
    end loop;
    if s.debug then put_line ("bspace="); put (bspace); new_line;
    end if;
    for move in up..right loop
        state := cur.state;
        case move is
            when up => new_pos := bspace - size;
                       valid := bspace > 3;
            when down => new_pos := bspace + size;
                       valid := bspace < 7;

            when left => new_pos := bspace - 1;
                       valid := bspace mod 3 /= 1;
            when right => new_pos := bspace + 1;
                       valid := bspace mod 3 /= 0;
        end case;
        if valid then
            state(bspace) := state (new_pos);
            state(new_pos) := blank;
            if state /= cur.parent.state then
                generate_new_sg_node (dn,s,state);
            end if;
        end if;
    end loop;
    s.x := s.x + 1;
    put (s.x,5);
    if debug then put_line ("leaving expand");
    end if;
end expand;
```

APPENDIX D. EXPAND for Flight Path Generator

```
procedure expand (s : in out sys_rec_type; dn : in out dnode_type;
                dom : in out domain_type) is
 -- generate legs

i : integer;
leg_angle : float;
new_point, point : statetype;
begin
    if s.debug then
        put_line ("entering expand");
        put (dn.current.state.x_coord);
        put (dn.current.state.y_coord);
        new_line;
    end if;
    point := dn.current.state;
    if distance (point, dn.dnode(dn.o).state) <= dom.leg_length then
        generate_new_sg_node (dn,s,dn.dnode(dn.o).state);
    else
        leg_angle := - (dom.path_arc/2.0) - dom.leg_arc;
        for i in 1..dom.legs_per_arc loop
            leg_angle := leg_angle + dom.leg_arc;
            if dn.dnode(dn.o).state.x_coord > point.x_coord then
                new_point.x_coord := point.x_coord +
                                    dom.leg_length*cos(leg_angle);
            else
                new_point.x_coord := point.x_coord -
                                    dom.leg_length*cos(leg_angle);
            end if;
            new_point.x_coord := five_signif (new_point.x_coord);
            new_point.y_coord := point.y_coord +
                                    dom.leg_length * sin(leg_angle);
            new_point.y_coord := five_signif (new_point.y_coord);
            if s.debug then put (new_point.x_coord);
                            put (new_point.y_coord);
                            new_line;
                            put (s.goal.x_coord);
                            put (s.goal.y_coord);
            end if;

            -- is the new point within the boundaries?

            if new_point.y_coord < 0.0 or new_point.x_coord < 0.0 or
               new_point.y_coord > float(dom.y_limit) or
               new_point.x_coord > float(dom.x_limit) or
               (abs(atan((s.goal.y_coord - new_point.y_coord) /
               (s.goal.x_coord - new_point.x_coord))) >
               dom.path_arc / 2.0) then
                    null;
            else
                generate_new_sg_node (dn,s,new_point);
                if s.finished and dn.min_ptr(dn.d) /= null and
                                    dn.min_ptr(dn.o) /= null then
                    if dn.min_ptr(dn.d).state /= dn.min_ptr(dn.o).state the
                    generate_new_sg_node (dn,s,dn.dnode(dn.o).state);
                    end if;
                end if;
            end if;
        end loop;
    end if;
    s.x := s.x + 1;
    if s.x = 1000 then put_line ("1000");
    end if;
    if debug then
        put (s.x,5);
        put_line ("leaving expand");
    end if;
end expand;
```

THE GENERIC ARCHITECTURE APPROACH TO REUSABLE SOFTWARE

Gerald R. Brown
U.S. Army, CECOM

Richard B. Quanrud
SofTech, Inc.

ABSTRACT

Generic architectures provide an approach to the development of reusable software for families of related applications. They provide both a high level design and a set of reusable components to be used in the applications supported by the architecture. The components are typically larger and more complex and result in higher levels of software reuse than with conventional reusable components.

1 INTRODUCTION

Generic architectures provide an approach to the development of reusable software that is significantly different from the more conventional library of reusable components. A library typically provides a large number of components that are potentially reusable across a wide range of applications. The components of most libraries stand alone, i.e., they have few interdependencies. However, some libraries are designed for more restricted application domains in order to meet the particular requirements of those applications and improve the opportunities for reuse.

A generic architecture provides both a design and a set of reusable components for use in the development of applications within a specific application domain. It provides a much smaller number of components than normally found in a library, and most of those components are reused in each of the applications of the domain. The components tend to be larger, more complex, and are highly integrated with the other components provided by the architecture.

With a generic architecture, generality is sacrificed to obtain a higher level of software reuse than is normally possible with library components. The components are both application and design specific and therefore incorporate more assumptions about their intended use than is possible with more general purpose components. This allows the components to be both larger and more complex than would be the case without such assumptions.

A second objective is adaptability. Within the domain of the architecture, the design and the components must be adaptable to the needs of individual applications. As a result, applications based on a generic architecture are inherently more adaptable to changing requirements, an important consideration in the development of any software with a long life cycle.

A number of different programming languages can be used to implement a generic architecture. The most appropriate are probably those with strong object oriented features. The language choice can have an important impact on the design and on the implementation of the components. This discussion is limited to generic architectures that are implemented in Ada. Ada has important strengths as well as some weaknesses in this area. An objective of the current effort is to identify the most effective implementation techniques for use with that language.

The basic concepts of a generic architecture are not new. Dijkstra[3] in his original paper on "Structured Programming" discussed "incomplete programs" constructed from "programming pearls" or reusable modules. Parnas[4] described techniques for developing and using reusable modules in the development of programs belonging to "program families". Similar ideas can be found in reconfigurable operating systems, application generators, and particularly in more recent attempts to develop application frameworks such as the MacApp[tm] package for the Apple Macintosh[tm]. SofTech has used the generic architecture approach in the development of software for the RAPID* and RAPIER** projects.

A generic architectures is not intended for use outside of its specific domain. The expectation is that a separate architecture will be needed for each different application domain. Each generic architecture has a domain, a design, and a set of reusable components which are specific to that architecture. The material that follows discusses domains, designs, and reusable components in greater detail.

*Reusable Ada Packages for Information Systems Development

**Rapid Emergency Reconstitution System

2 THE DOMAIN

The domain limits the generality required in the design and in the reusable components so as to set bounds on the complexity of both. This, in turn, limits the effort required to implement the architecture. The domain also identifies the applications that can be supported by the architecture. This provides a basis for estimating the implementation effort and the size of the domain that can be used in a cost/benefit analysis.

The selection of an appropriate domain is critical to the success of a generic architecture. Applications within the scope of the domain should be similar enough to share a common design and a significant number of components. It helps if they also share a common hardware and software environment, although the importance of environmental dependencies can be limited by confining them to a small number of components.

It is also highly desirable that the applications within the domain be under the control of a single organization. That organization is the best source of financial support. It can also assist in the resolution of issues, support the use of the architecture in the development of applications, and reap the benefits of this approach to software development.

A domain analysis is conducted to establish the scope of the domain. It should identify the requirements that are common to the applications of the domain and include the cost/benefit analysis that is needed to justify the up front effort required to develop the architecture. The development of a generic architecture is appropriate only if the number of applications in the domain is large enough to justify the development effort.

The domain analysis must also include the development of the preliminary design for the architecture. The design must be specified to some level of detail in order to know whether it can be shared by all of the applications within the scope of the domain. Applications that cannot share a common design cannot be included in the same domain. Thus, the design plays a critical role in determining the scope of the domain itself.

3 THE DESIGN

The high level design must meet the requirements of all of the applications in the domain of the architecture. Application specific adaptations of the design are possible at lower levels, but the overall flow of control should be the same for all applications. This is because the components are likely to contain design dependencies that may affect their behavior in a different design context.

The design normally identifies a complete set of components for a basic application that is developed from the architecture. These are typically the only components that are provided with a generic architecture. They are highly integrated and interact with each other in ways dictated by the design. If a component is removed, it must usually be replaced by a component that fills the same role. However, at least some of the components must be adapted to the requirements of an application and some applications may require components that were not included with the architecture. Thus, a generic architecture may be treated as an incomplete, but adaptable, application.

In discussing the decomposition of a system into modules, Parnas[5] proposed that the designer begin by listing design decisions which are difficult or likely to change. "Each module is then designed to hide such a decision from others." This is a particularly appropriate guideline for the isolation of dependencies in the design of a generic architecture.

The design should isolate hardware and software dependencies by confining each type of dependency to a small number of components. However, it should be recognized that there are several distinctly different types of hardware and software dependencies. Hardware dependencies range from those related to the instruction set architecture to dependencies on specific display or other input-output device characteristics. Software dependencies may be associated with specific software subsystems such as an operating system, a data base management system, or a graphics package. The use of separate components to encapsulate each type of dependency provides added flexibility in dealing with later changes.

Application dependencies result from the unique requirements of each of the applications in the domain. They are likely to be more widespread and harder to isolate than those associated with the hardware/software environment. Components with application dependencies should be adaptable to the types of changes needed to meet the specific requirements of the applications. Dummy components are often used as place holders for components that will contain large amounts of application unique code.

There are at least two reasonable approaches to the development of a design. The first is to examine the design of existing applications within the same domain. A great deal can be learned from such an analysis even if the existing designs are not reused with the architecture. The second approach is to develop a prototype architecture and prototype applications as a way to test a proposed design. The development of prototype applications that use the prototype architecture is necessary to validate the design.

Use of the principles of object-oriented design is also highly desirable. An object orientation provides a basis for partitioning the elements of the design into reusable components. Object-oriented components tend to be inherently reusable. This is because they tend to make fewer references to data that is defined outside the using component. Finally, an object orientation provides an easily understood role for each component, a feature which facilitates both the use and maintenance of the components. Booch[2], Schmucker[6], Seidewitz[7], and Berard[1] are good sources for additional information on object-oriented techniques.

4 THE REUSABLE COMPONENTS

A reusable component of a generic architecture should represent an abstract object and encapsulate all of the data and operations required to manipulate the abstract representation of that object. In Ada terms, a component will usually be an Ada package, the data representation of the object will be private to the package, and all of the operations that may be invoked from outside to manipulate that data representation will be identified as subprograms in the package specification.

The reusable components provided by a generic architecture are fundamentally different from those that might be found in a library of reusable components. They tend to be larger and have complex interrelationships with the other components provided by the architecture. Most of the components invoke operations of other components provided by the architecture and depend on those components for essential services. A component typically encapsulates a large number of separate operations on the abstract object supported by the component. The number and types of operations are dictated by the requirements of the domain and the design provided by the architecture.

An Example

Assume that a number of workstations are linked through a communications network as shown in Figure 1. Different applications are executed on each of the workstations, but all of the applications are within the domain of a single generic architecture. Messages are exchanged among the applications, but the messages produced by each application are different from those produced by the other applications.

One of the principal reusable components provided by a generic architecture for those applications would be a message manager. The object orientation of the component is to messages. The component encapsulates all of the data required to represent the messages exchanged by the applications and all of the operations that are performed on messages. A partial list of the operations that might be performed on a message is contained in Table 1. The actual number of operations that might be supported by the component is likely to be in the range of 50-100, if allowance is made for different types of messages, housekeeping and initialization operations, etc. The component is large and tailored specifically to the requirements of this network of applications.

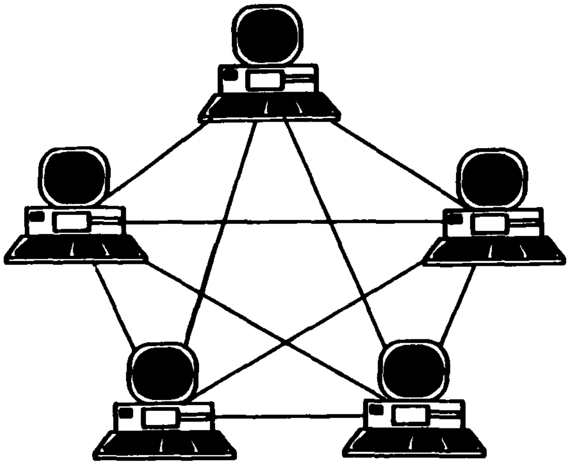


Figure 1. The Workstation Network

Table 1. Partial List of Operations Performed by the Message Manager

Operation	Purpose
Create	Initializes data representation
Address	Assigns one or more addresses
Compose	Composes the text of the message
Edit	Supports operator editing
Display	Displays message to the operator
Classify	Assigns security classification
Send	Submits message for transmission
Receive	Accepts a message from the queue
Reply	Assembles a reply
Forward	Forwards to a new addressee
Print	Prints a hard copy
File	Files for later retrieval
Archive	Records on the archive device
Delete	Deletes the data representation

The message-manager component does not contain all of the code needed to carry out its operations. A large share of the operations require support from other components provided by the architecture. Table 2 provides a partial list of the message-manager operations that invoke the operations of other components and the type of support that is provided in each case. Again the table shows only a small share of those outside requests. When all such requests are considered, the message manager is shown to have a complex set of dependencies on the other components provided by the architecture.

Table 2. Partial List of Services Provided by Other Components

Message Operation Invoked Operation (Invoked Component)	Service Provided
Address	
Get Distribution (Data Base Mgr.)	The standard list of message recipients
Get Address (Data Base Mgr.)	The current address of each recipient
Compose	
Get Operator Input (Dialog Mgr.)	Message parameters and other operator input
Get Template (Data Base Mgr.)	The text string used to compose the message
Insert Text (String Mgr.)	Insertion of text into the message template
Print	
Queue for Printing (Printer Mgr.)	Staging of the message for printing

Adaptation Techniques

Components that do not meet the specific requirements of an application must either be adapted to those requirements or replaced. The most important mechanism for the adaptation of an object-oriented component is inheritance. Inheritance allows a component to inherit the data representations and operations of an existing component, add data to the data representations, and add or replace operations on the data.

Ada provides no real support for inheritance. The use of derived types allows one to export a data representation from one package to another along with the operations on that data representation. However, the data representation cannot be private. The exposure of the details of the representation to one package provides those details to any other package that may want to perform operations on that data. This undermines the integrity of the original component as it no longer encapsulates all of the operations on the data. The data is now global to the system; side effects are harder to control; and reusability has been compromised.

The most useful Ada mechanisms for the adaptation of a component are generics and separate subunits. A component represented by a generic package can be adapted through the substitution of type, value, object, and subprogram declarations contained within the package. The specific declarations that may be substituted are identified as generic parameters in the original generic package.

Separate subunits allow the implementation of an operation to be provided as a separate subprogram. That subprogram can then be replaced without changing the reusable component itself. However, the code in the component's package body must indicate that the subprogram is being provided separately.

The main problem with both of these techniques is that the specific types of adaptations that may be made to a component must be anticipated in advance. This requires additional effort during the domain analysis and design stages to identify the types of adaptations that may be required. They also provide no way to add new operations or data to an existing component.

A component can always be replaced when it cannot be adapted to meet the requirements of the current application. This may not require a great deal of effort because much of the code from the original component can probably be salvaged. However, it can result in a second version of a rather large component, with a commensurate increase in the longer term maintenance burden.

Adaptation requirements have a major impact on the documentation for a generic architecture. With a library of reusable components, the documentation assists the user in the selection of the appropriate component from a library containing a large number of components. With a generic architecture, the user starts with the assumption that each component will either be used or replaced. The documentation deals with the role of the component within the architecture and the adaptation of the component to meet the requirements of specific applications.

5 CONCLUSIONS

Within a given application, higher levels of software reuse are achievable through the use of a generic architecture than with a more conventional library of reusable components. The components are larger and more complex because they are designed specifically to be used with the design and to meet many of the requirements of the application. Most of the components of such an application are likely to have been provided by the architecture.

Because the level of reuse is high, both development and maintenance costs should be substantially lower than with other development approaches. In addition, the emphasis on the adaptability of the design and components to meet the needs of specific applications leads to applications that are more adaptable to future changes in requirements. The underlying adaptable structure is present in each of the applications just as it is in the architecture itself. Documentation needed to adapt the architecture to new applications provides most of the information needed to change or maintain those applications in the future.

Interoperability is likely to be enhanced among the applications that use the same generic architecture. Message protocols and other standards can be supported more consistently across applications if they are implemented through common components. A consistent man-machine interface, that results from the use of common graphics routines, can reduce training costs and allow operators to be moved more easily from one application to another.

Rapid prototyping is facilitated by a generic architecture because the design and most of the code is already in place. The adaptability of the code supports the development of the final application through a series of incremental refinements, with the opportunity for user experimentation and reaction at each step in the process.

However, because the components are application and design specific, they are likely to be useful only in the domain for which they were developed. Moreover, the cost of development must be justified solely on the basis of the contribution they make to the applications of that domain.

Generic architectures are intended to produce higher levels of software reuse. However, they also represent good software engineering practice. They help to clarify the specific objectives of a project with respect to reusability and adaptability. This in turn leads to software that is adaptable, improved interoperability among applications, and support for early prototyping and incremental refinement of new applications. In effect, the approach formalizes many of the practices of a well run project team.

REFERENCES

1. E.V. Berard, An Object-Oriented Handbook for Ada Software, E.V.B. Software Engineering, Inc., 1985

2. G. Booch, "Object-Oriented Development." IEEE Transactions on Software Engineering, vol. SE-12, no. 2, p. 211, February 1986.

3. E. Dijkstra, "Structured Programming" Software Engineering Techniques, Report on a Conference Sponsored by the NATO Science Committee, p. 84, October 1969.

4. D.L. Parnas, "On the Design and Development of Program Families." IEEE Transactions on Software Engineering, vol. SE-2, no. 1, p. 1, March 1976.

5. D.L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules." Communications of the ACM, vol. 5, no. 12, p. 1053, December 1972.

6. K.J. Schmucker. Object-Oriented Programming for the Macintosh[tm], Hayden Book Company, 1986.

7. E. Seidewitz and M. Stark. General Object-Oriented Software Development, Goddard Space Flight Center, August 1986.

Gerald R. Brown is an electronics engineer with the U.S. Army at Ft. Monmouth, NJ. He is responsible for the development and demonstration of reusable software technology within the Center for Software Engineering.

Richard B. Quanrud is a principal investigator with SofTech, Inc. in Waltham, MA. He is currently interested in the software development process and its application to the development of command and control systems.

Searching for Reusable Software Components with the RAPID Center Library System

Ernesto Guerrieri
SofTech, Inc., Waltham, Massachusetts 02254-9197

Abstract

This paper describes the SIDPERS-3 RAPID Center project that SofTech, Inc. has undertaken with the U.S. Army Information System Engineering Command (ISEC). It describes the development of the SIDPERS-3 RAPID Center for promoting the successful reuse of software and, in particular, the development of the RAPID Center Library (RCL) System as an automated tool for the identification and retrieval of Reusable Software Components (RSCs) from the RAPID Library.

Introduction

The success of software reusability depends on several factors. The Reusable Software Components (RSCs) need to be of high quality and very reliable so that other software development projects can seriously consider their use. There needs to be a simple and fast way to identify the most appropriate RSC for another software development project. Furthermore, the integration of the selected RSCs into another software system should be easy.

As the number of RSCs available grows, it will be important to the success of software reusability that the above goals be achieved in a coherent and organized manner (i.e., through proper management of RSCs). These are some of the objectives of the SIDPERS-3 RAPID Center project that SofTech, Inc. has undertaken with the U.S. Army Information System Engineering Command (ISEC).

This paper describes the development of the SIDPERS-3 RAPID Center for promoting the successful reuse of software and, in particular, the development of the RAPID Center Library (RCL) system as an automated tool for the identification and retrieval of RSCs in the RAPID library.

The RAPID Center

The RAPID Center will be a support center for software reuse, providing expert assistance and sophisticated tools to SIDPERS-3[1] engineers. The RAPID Center concept derived from the findings of the ISEC RAPID[2] investigation performed by SofTech [RUEGSEGGER87] for ISEC. The RAPID Center is currently being developed by the US Army and SofTech. The project is funded by the Software Technology for Reliable Adaptable Systems (STARS) program.

The overall objective of the RAPID Center project is to provide SIDPERS, and ultimately ISEC, with a significant and meaningful aid to developing Ada programs through reusability.

The tasks involved include the development of:

- a comprehensive plan for the development of the SIDPERS-3 RAPID Center. This will include:
  - RAPID policy recommendations,

[1] SIDPERS-3: Standard Installation Division Personnel System, version 3

[2] RAPID: Reusable Ada Packages for Information system Development

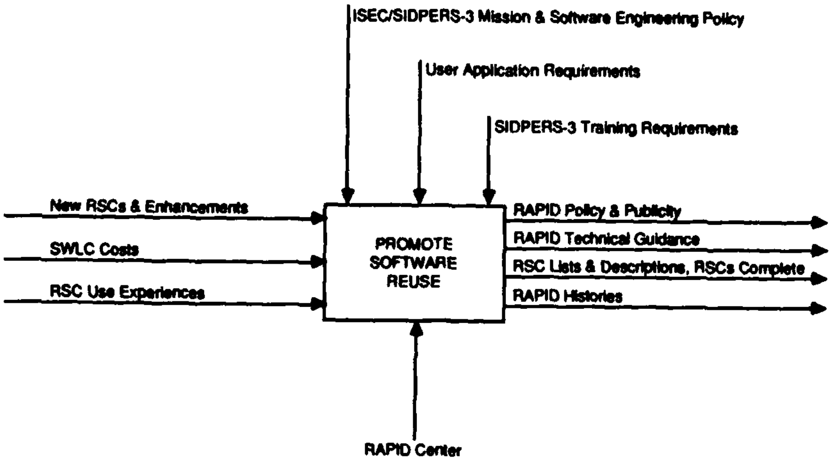

ISEC/SIDPERS-3 Mission & Software Engineering Policy
User Application Requirements
SIDPERS-3 Training Requirements
New RSCs & Enhancements
SWLC Costs
RSC Use Experiences
PROMOTE SOFTWARE REUSE
RAPID Policy & Publicity
RAPID Technical Guidance
RSC Lists & Descriptions, RSCs Complete
RAPID Histories
RAPID Center

Figure 1. SADT diagram "Promote Software Reuse" (Context).

- RAPID system administration guidelines,
- RAPID programmer/user guidelines, and
- RAPID reusability standards.

- the RAPID Center Library system, to provide SIDPERS-3 personnel with a fast effective means of locating and identifying RSCs so that it will be more effective for the programmer to reuse software than to develop the software himself.
- ten RSCs, to provide SIDPERS-3 personnel with case studies of reusable software components shown to be usable within SIDPERS-3.

The SIDPERS-3 RAPID Center is conceived as a center of excellence on Ada reusability issues and provides technical services in promoting reusability within ISEC. The primary activity of the Center is not simply furnishing and maintaining reusable software, but the broader mission of promoting the successful reuse of software throughout SIDPERS-3 and, ultimately, ISEC. The main goal of the RAPID Center is to **promote software reuse** (see Figure 1). This is achieved via the issuance of RAPID policy and publicity (including policy, recommendations, reuse advocacy, and solicitations for user feedback). The RAPID Center performs four principal functions in accomplishing its comprehensive activity of *promoting software reuse* (see Figure 2):

1. Develop and Maintain RSCs,
2. Catalog and Retrieve RSCs,
3. Provide Technical Guidance, and
4. Analyze and Exploit Experience.

Develop and Maintain RSCs

The RAPID Center staff, guided by the recommended RAPID policy and the RAPID reusability standards, ensure that the software components submitted to the library are of acceptable quality, complete, and properly documented. Some new RSCs and enhancements are submitted from outside the Center; the staff complete them as needed and test them. Other components are developed by the staff in response to recommendations for new RSCs and enhancements. Bug reports are analyzed and problems corrected; change notices are distributed to RAPID users. The primary output of this activity is library RSC packages (see Figure 2, box 1). Such packages include the complete component as the user will receive them, along with documentation, test results, and suitability assessments used by the library to aid the user.

The principal activities include:

- the development/maintenance of RSCs,
- the acquisition of RSCs, and
- the acceptance and validation of RSCs.

Catalog and Retrieve RSCs

Once an RSC is accepted into the RAPID Center Library, it is classified and cataloged (similar to what is done in a "book" library) for retrieval purposes. The classification scheme adopted is a faceted classification scheme [PRIETO-DIAZ87a]. With a faceted classification scheme, a software component is described by a set of software component attributes (i.e., facets) that are assigned values (i.e., facet terms).

The RAPID Center Library system aids a user of the RAPID Center to identify and retrieve the most suitable RSC in the RAPID Library (see Figure 2, box 2). The user enters a description of the desired software component and the RAPID Center Library System proposes possible candidate RSCs that are available through the RAPID Library. A suitability assessment is made to see how well each of the candidate RSCs match the user's desired software component. Based on the suitability assessment and other data (e.g., RSC description, RSC use history, bug reports, documentation, software metrics, etc.), the user can narrow the choice to the most suitable RSC. When the user has narrowed the choice sufficiently, the system provides everything the user needs to know to obtain the component source code and all documentation. The source code for most components is obtained directly from the RAPID Library, though the system may well identify components in some other library or that are available commercially. The system compiles an RSC use log and a log of search failures that will aid in monitoring the success of the RAPID Center.

Comparable library systems can be found in [BURTON87], [KATZ87], and [PRIETO-DIAZ87b].

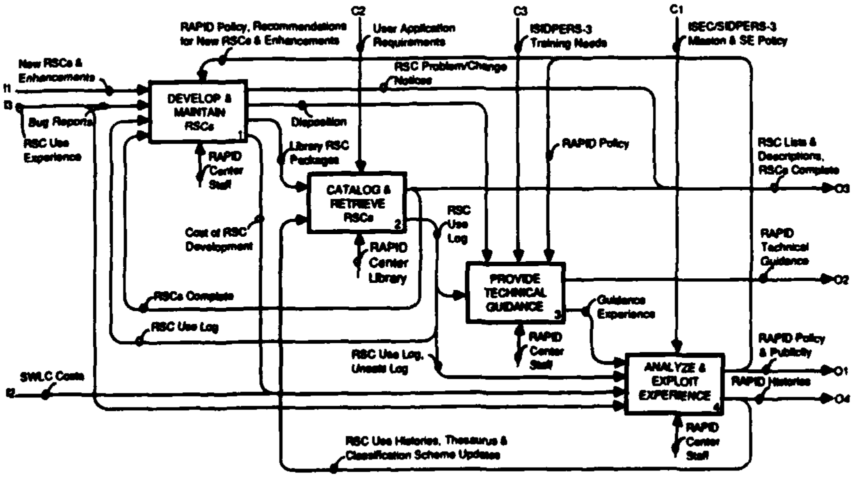


Figure 2. SADT diagram "Promote Software Reuse."

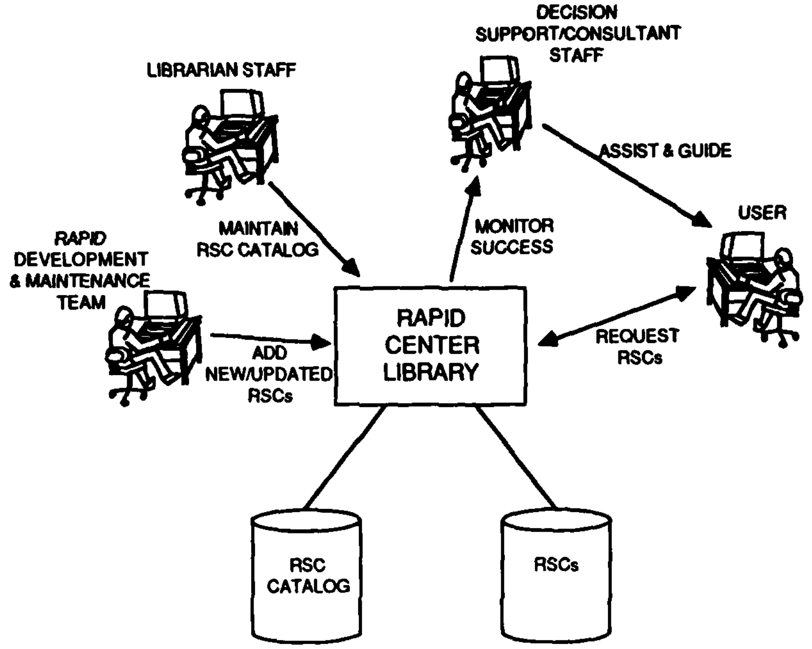


Figure 3. The RAPID Center Library System.

Provide Technical Guidance

The RAPID Center staff helps the SIDPERS-3 software engineers identify and develop RSCs from ongoing software developments, search for and request RSCs from the RAPID Center Library System, and integrate RSCs into new software developments (see Figure 2, box 3).

Analyze and Exploit Experience

To support the primary mission of promoting software reuse (via the primary RAPID policy and publicity), the staff tracks the Center's experience in running the library system (via RSC use, search failures) and in guiding users (via consulting), and tracks the RAPID user's experiences with the RSCs that they have drawn from the library (see Figure 2, box 4). The staff also collects costs from inside and outside the Center to compare software reuse with conventional software development practices. Much of this data, analyzed and condensed, is provided to managers as RAPID histories *for decision support*. Those histories pertaining to the use of individual RSCs are fed back into the library system as information to help users evaluate RSCs. Search failure information results in recommendations for new or enhanced RSCs, or in updates to RAPID Center Library search mechanisms (i.e., the classification scheme and the underlying thesaurus), depending on the cause of the search failure.

The RAPID Center Library System

The RAPID Center Library (RCL) System is a software system designed for the automated cataloging and retrieval of RSCs in the SIDPERS-3 RAPID Center Library so that users can **find suitable RSCs**. The system is being implemented in Ada for a Sperry 7080 with Unix[3] System V operating system. The software is planned to be operational in August 1988.

[3] Unix is a trademark of AT&T Bell Laboratories.

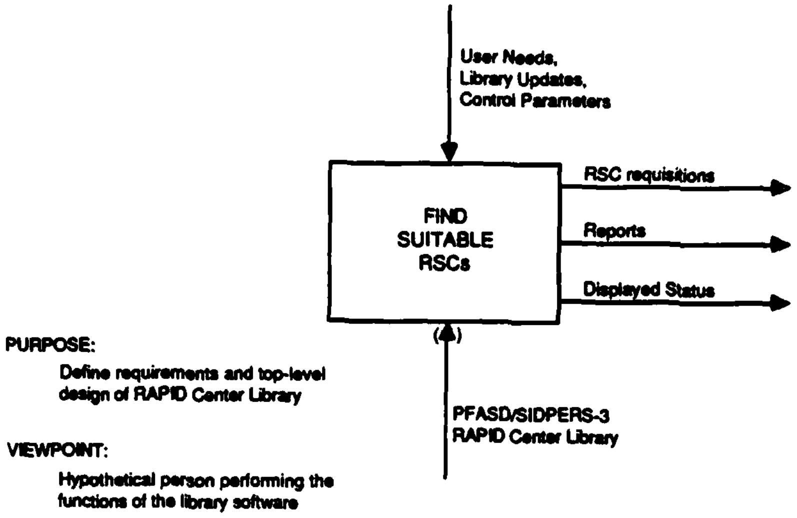


Figure 4. SADT diagram "Find Suitable RSCs" (Context).

The RAPID Center's goal is to promote software reuse. The RCL system supports the goal of software reuse in two ways. First, it provides the RAPID user (typically a software engineer), with an automated and user-friendly tool for searching for a reusable software component. Second, it leads the RAPID user to one or more components particularly suited to his needs. Both of these services add validity and appeal to the Center's role in publicizing and promoting software reuse.

The RAPID user searches for and obtains an appropriate software component to reuse via the RCL (see Figure 3). The RAPID decision support and consulting staff monitor the RCL success in satisfying its users, in order to fine-tune the system (recommending new/enhanced RSCs or refining the RCL searching mechanism) and give guidance and technical assistance to the RAPID users. The RAPID librarians maintain the RSC Catalog and operate the system. The RAPID development and maintenance team incorporate the new/updated RSCs into the library.

The major functions of the RCL system are (see Figure 4 & 5) to:

- identify an RSC,
- extract an RSC,
- maintain the search apparatus,
- maintain the RSC library catalog,
- generate reports, and
- control system actions.

There are two principal classes of users of the RCL system: the *RAPID user* that searches for and requests a copy of an RSC, and the *RAPID librarian* that maintains the system (i.e., the search apparatus and the RSC library catalog) and generates reports.

RSC Identification

Prior to requesting an RSC, the RAPID user needs to identify an RSC in the RAPID library that best meets the needs of the user. The RSC identification process will allow the user to describe the requirements of the desired software component, search the RSC catalog database for candidate RSCs which closely match the user's requirements, assess the

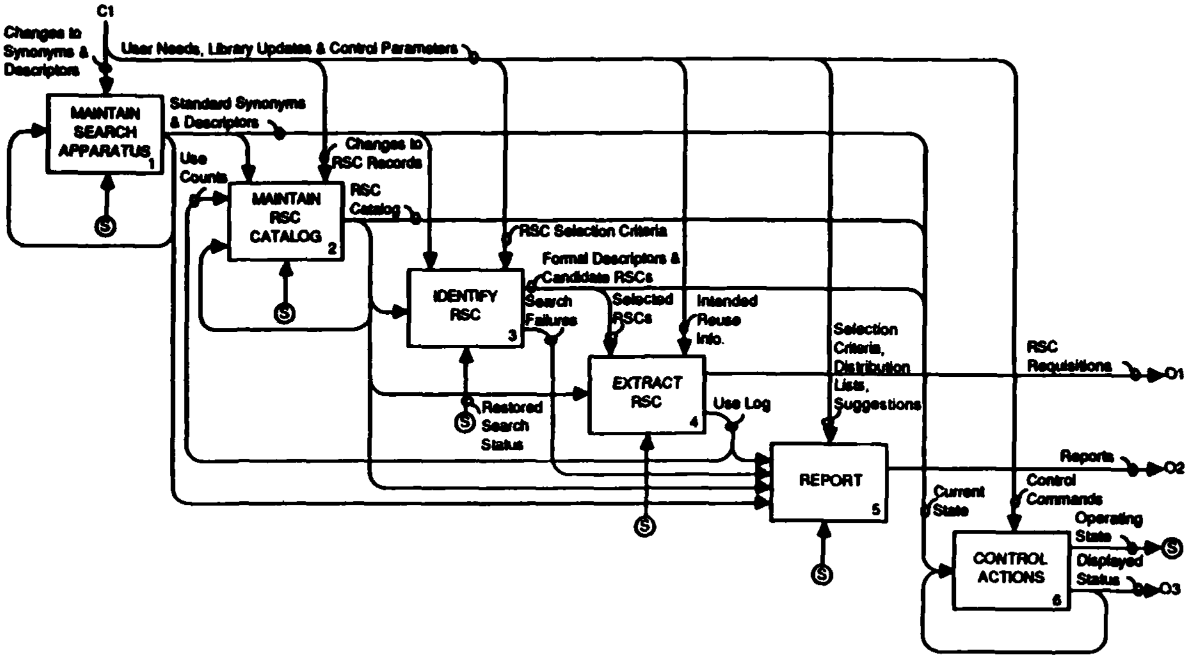


Figure 5. SADT diagram "Find Suitable RSCs."

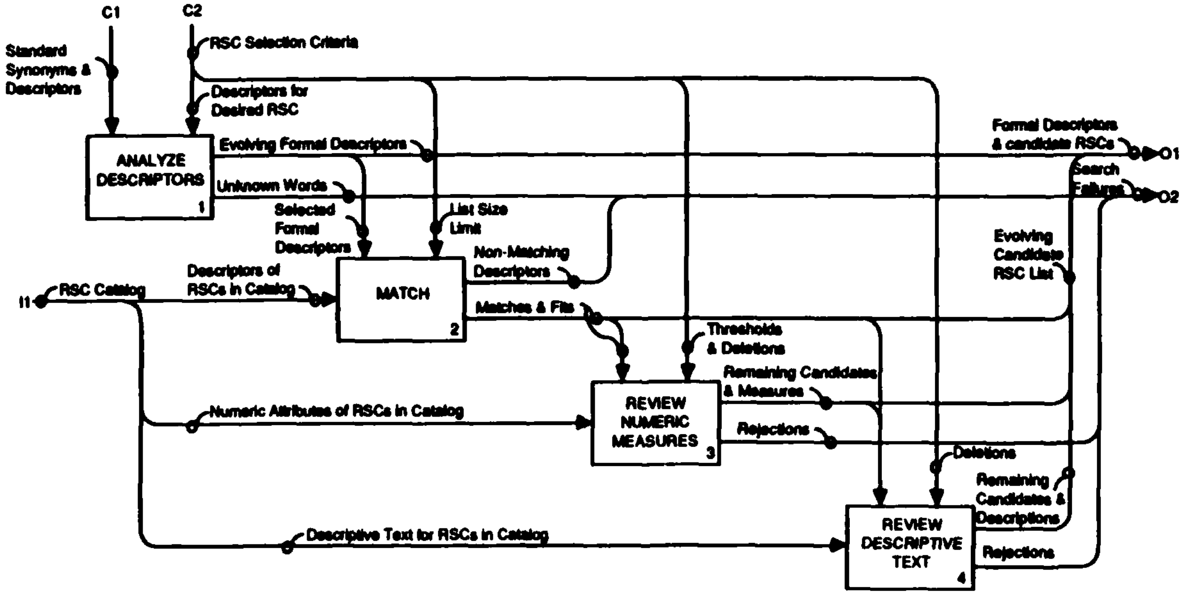

C1
C2
Standard Synonyms & Descriptors
RSC Selection Criteria
Descriptors for Desired RSC
ANALYZE DESCRIPTORS 1
Evolving Formal Descriptors
Formal Descriptors & candidate RSCs
O1
Unknown Words
Search Failures
O2
Selected Formal Descriptors
List Size Limit
Non-Matching Descriptors
Evolving Candidate RSC List
I1
RSC Catalog
Descriptors of RSCs in Catalog
MATCH 2
Matches & Fits
Thresholds & Deletions
Remaining Candidates & Measures
Numeric Attributes of RSCs in Catalog
REVIEW NUMERIC MEASURES 3
Rejections
Deletions
Remaining Candidates & Descriptions
Descriptive Text for RSCs in Catalog
REVIEW DESCRIPTIVE TEXT 4
Rejections

Figure 6. SADT diagram "Identify RSC."

suitability of the candidate RSCs according to the user's requirements, display available on-line information for each of the candidate RSCs, and refine the list of candidate RSCs. This process can be repeated until the user has identified a suitable RSC.

The major activities in RSC identification are (see Figure 6):

- request an RSC (analyze descriptors),
- search for candidate RSCs (match),
- browse through candidate RSC information (review numeric measures and review descriptive text), and
- refine RSC candidate list (review numeric measures and review descriptive text).

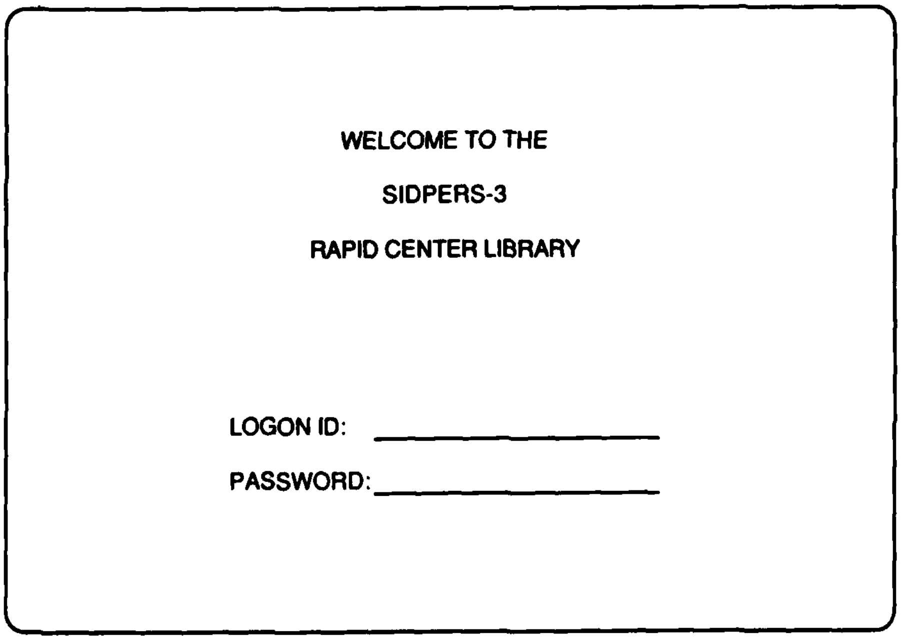
RSC Request

The RAPID user enters the requirements for the desired software component as a description of the software component (see Figure 8). A software component is described by a set of software descriptors. Each software descriptor describes an attribute (i.e., aspect or facet) of the software component. For example:

> (FUNCTION:sort, METHOD:exchange,
> OBJECT:integer, . . .)

describes[4] a software component that sorts integers using the *exchange* method. At any point, the user can ask the system for a list of known valid choices (for the attribute-name or attribute-value fields) from which the user can choose.

Once the user has input the description of the software component, the system analyzes the input and classifies it according to the RSC taxonomy (based on the RSC faceted



WELCOME TO THE

SIDPERS-3

RAPID CENTER LIBRARY

LOGON ID: ____________________

PASSWORD: ____________________

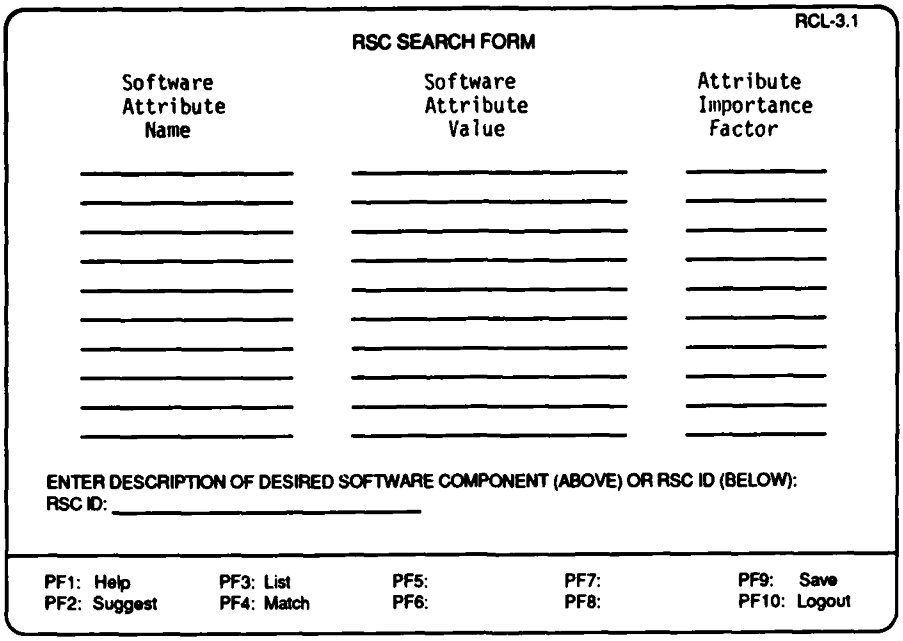
Figure 7. RAPID Center Library System login screen.

[4] A software description is represented as the set

(software-descriptor, . . .)

A software descriptor is represented as the pair

attribute-name attribute-value



RCL-3.1

RSC SEARCH FORM

Software Attribute Name	Software Attribute Value	Attribute Importance Factor

ENTER DESCRIPTION OF DESIRED SOFTWARE COMPONENT (ABOVE) OR RSC ID (BELOW):
RSC ID: ____________________

PF1: Help	PF3: List	PF5:	PF7:	PF9: Save
PF2: Suggest	PF4: Match	PF6:	PF8:	PF10: Logout

Figure 8. RAPID Center Library System RSC search screen.

classification scheme and thesaurus). The analysis produces a formal software component description which is later used to search the RSC catalog. If the analysis encounters an unknown descriptor, the system queries the user for clarifications until a valid description is obtained.

RSC Catalog Search

The search for Candidate RSCs function matches the formal software component description of the software component requested by the user to the RSCs in the RAPID library catalog. Given a formal software component description, a query will be made to find the RSC descriptions that match the formal software component description. It will utilize the importance factor (or priority index) set by the user to guide in the search.

Suitability Assessment

Once one or more RSC descriptions has been matched to the formal software component description, the RSCs are ranked according to the user's importance factor and a suitability assessment. The suitability assessment gives an index on the closeness of the match between the user description of the software component desired and the description used to classify the chosen RSC in the RAPID library. The system allows for inexact matches of RSCs to occur for two primary reasons. First, it allows the user to evaluate RSCs that would be excluded in exact matches but that may be easily modified for the user's purpose[5]. Second, it allows the user to search even though an inexact description was given.

Browsing Capability

Once a list of candidate RSCs has been constructed (the RSC Candidate List), the user can examine the on-line in-

[5] This is useful when the system does not make an exact match

formation available for each of the RSCs. The RSC information is divided into textual and numeric information. The textual information consists of on-line documentation, abstract, summary sheet, bug report summary, etc. and can be browsed or printed. The numeric information consists of software measures (i.e., lines of code, reusability index, number of reuses, software metrics, etc.) [GUERRIERI87] that can be used as comparative data, thus allowing a user to compare similar values across selected RSCs.

Refinement Process

The main objective of the user is to find and retrieve an RSC that satisfies the description of the desired software component. Given a list of the candidate RSCs, the user needs to choose the most suitable one. The system has made an attempt to rank these RSCs according to the suitability assessment, but further refinement can be made after the user has browsed through the on-line information of the RSCs. The user can eliminate those RSC from the Candidate RSC List that are inappropriate or do not satisfy some criteria (which may have been omitted during the initial component description).

The user can also refine the Candidate RSC List by refining the description of the desired software component. While browsing, the user may notice other descriptive terms in the documentation of an RSC that may better describe the desired component. This can either narrow the search or correct the search direction.

RSC Extraction

Once the user has refined the search to the point in which the most suitable RSC has been identified, the user can request for a copy of the RSC complete. If the RSC is directly available from the RAPID Library, a copy of the RSC complete will be directed to the medium specified by the user (i.e., user directory space, magnetic tape, diskettes, modem, etc.). If the RSC is not directly available, the system may identify a software component in some other library or that is available commercially. Appropriate instructions for retrieving the RSC would be issued.

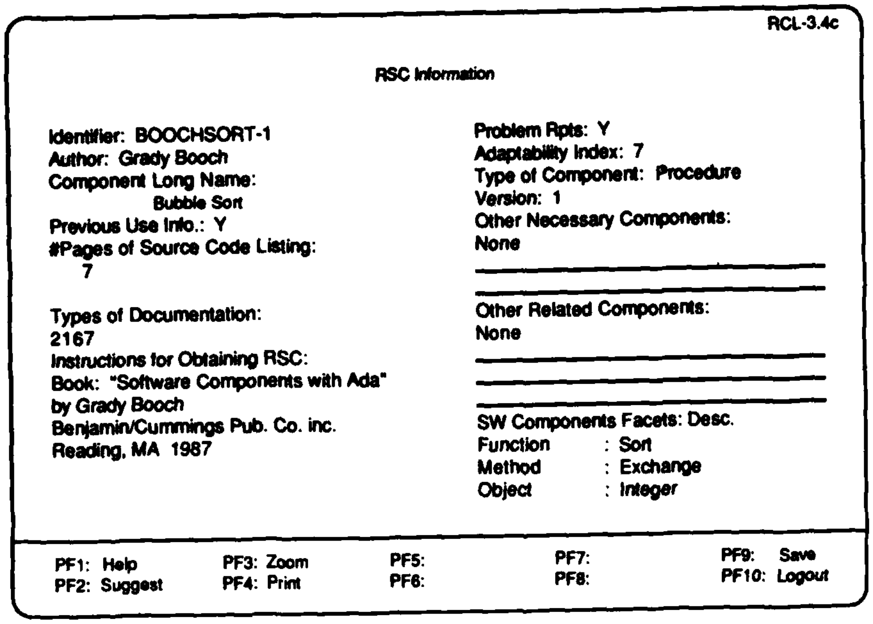
Search Mechanism Maintenance

In order for the RAPID Center Library system to be successful, it must inspire confidence among its users that their search will be worth their time and money. As experienced is gained from past usage, the system will be updated to facilitate the identification process. This will include fine tuning of the search mechanism as well as enriching the RAPID Library with commonly requested RSCs.

Fine tuning of the search mechanism is achieved by maintaining the search mechanism: the classification scheme and the thesaurus. Each RSC is given a description according to a faceted classification scheme; a version of which is on-line. As the categorization of software is refined, the classification scheme needs to be updated to reflect the refinements. Since the user's terminology will differ from the terminology utilized by the classification scheme, there is also a need to bridge the gap between the formal terminology used by the system and the user's terminology. This is achieved through a thesaurus. When a new term is utilized within the system, a dialog between the system and the user tries to resolve the user's use of the term. This is logged (in the search failure log) as candidate refinements of the thesaurus.

RSC Catalog Maintenance

The RAPID Center Library system needs to be kept updated so that the new or updated RSCs can be searched within a realistic time frame. This can be accomplished by having provided an easy mechanism for the librarians to update, add, or delete RSCs from the RSC Catalog. Updating an RSC may consist of a total update (i.e., replace RSC with a new version), or a partial update (i.e., abstract, bug report



RCL-3.4c

RSC Information

Identifier: BOOCHSORT-1
Author: Grady Booch
Component Long Name:
Bubble Sort
Previous Use Info.: Y
#Pages of Source Code Listing:
7

Types of Documentation:
2167
Instructions for Obtaining RSC:
Book: "Software Components with Ada"
by Grady Booch
Benjamin/Cummings Pub. Co. inc.
Reading, MA 1987

Problem Rpts: Y
Adaptability Index: 7
Type of Component: Procedure
Version: 1
Other Necessary Components:
None

Other Related Components:
None

SW Components Facets: Desc.
Function : Sort
Method : Exchange
Object : Integer

PF1: Help
PF2: Suggest
PF3: Zoom
PF4: Print
PF5:
PF6:
PF7:
PF8:
PF9: Save
PF10: Logout

Figure 9. RAPID Center Library System RSC information screen.

summary, description, etc.). The deletion of an RSC can either be an archival procedure or an actual deletion of the RSC from the RAPID library.

Library Monitoring

The RAPID Center Library system logs a variety of information for the purpose of tracking its performance, evaluating possible changes to enhance the system, and triggering RAPID Center activities. The information logged includes:

- RSC Use
- Search Failures
- Suggestion Box
- User Accounts

Some of the data is automatically incorporated into the RSC catalog (i.e., RSC use, user logins, etc.), while other information are available through reports that can be generated by the system.

RSC Use

As an RSC is retrieved by a user, it is logged in the *RSC Use Log*. This information allows other users to know of the popularity of a certain RSC, allows the librarians to notify the RSC users of revisions and updates to an RSC, allows the librarians to track success of an RSC in a software development.

Search Failures

During the identification and search process, information is recorded so as to provide data for fine-tuning the search mechanism and recommending the development or acquisition of new RSCs. It also gives an indication on the success of the system.

Suggestion Box

The RAPID user can submit comments and suggestions into an on-line *Suggestion Box*. This is reviewed periodically by the RAPID staff for enhancements to the system as well as indications on areas where assistance and guidance would be beneficial.

User Accounts

User information is also tracked so as to assist and inform users of changes to either extracted RSCs or RAPID Center procedures. It is also used to control access to the RSCs and the system.

Report Generation

There exists a series of canned reports that the librarian can generate from the above logs as well as from on-line information of the RSCs in the RAPID Library. These reports are used by the RAPID staff (i.e., decision support staff and consultant staff) to monitor the success of the RAPID Center Library system as well as suggest enhancements to the system or to the center. in general.

RSC Development

To initially populate the RAPID Center Library, a set of ten RSCs are being developed. These RSCs will be components that can be utilized directly on the SIDPERS-3 project. The components will either be developed or adapted from existing public domain or Commercial-Of-The-Shelf (COTS) software.

Conclusions

The software lifecycle needs to incorporate the role of software reuse in order to reduce development costs and increase software reliability. The U.S. Army Information System Engineering Command recognizes this fact. The SIDPERS-3 RAPID Center is a proof of concept that will be applied to the development of SIDPERS-3. The current version will support only software components that are Ada code. Since the benefits of reuse can be achieved early in the life-cycle, future versions should be extended to include not only code, but also design, specifications, and documentation.

References

BURTON87 "The Reusable Software Library," Burton. B.A.. Aragon, R.W.. Bailey, S.A.. Koehler. K.D.. and L.A. Mayes, **IEEE Software**, Vol.4, No.4. pp.25-33, July 1987.

GUERRIERI87 "On Classification Schemes and Reusability Measurements for Reusable Software Components," Guerrieri, E., SofTech Technical Report TP-256, SofTech, Inc., 460 Totten Pond Road, Waltham, MA 02254-9197. October 1987. Also to be published in Workshop on Software Reuse Proceedings, Rocky Mountain Institute of Software Engineering.

KATZ87 "PARIS: A System for Reusing Partially Interpreted Schemas," Katz, S., Richter. C.A., and T. Khe-Sing. MCC Technical Report STP-026-87. Microelectronics and Computer Technology Corp., 9430 Research Blvd., Austin. Texas 78759-6509, January 30. 1987

PRIETO-DIAZ87a "Classifying Software for Reusability," Prieto-Diaz. R.. and P. Freeman, **IEEE Software**. Vol.4. No.1. pp.6-16. January 1987

PRIETO-DIAZ87b "Breathing New Life into Old Software." Prieto-Diaz, R., and G. Jones. **GTE Journal**, Vol.1, No.1. pp.22-31. June 1987

RUEGSEGGER87 "RAPID: Reusable Ada Packages for Information System Development," Ruegsegger. T., Technology Strategies '87 Proceedings. January. 1987

SOFTECH87 "SIDPERS-3 RAPID Center Policy Recommendation - Initial draft," SofTech, Inc., Contract No. 3451-4-112/5, SofTech, Inc., 460 Totten Pond Road, Waltham, MA 02254-9197, December 18, 1987

Biography

Ernesto Guerrieri
SofTech Inc.,
460 Totten Pond Road,
Waltham MA 02254-9197

Ernesto Guerrieri is a System Consultant at SofTech, Inc. in the Military Applications Department. He is the technical leader in the development effort of the RAPID Center Library system. He also manages several contracts among which the support effort for the Ada Language Maintanence and the maintenance of the ACVC Implementers' Guide. He is pursuing a PhD at Rensselaer Politechnic Institute in the area of software portability. His interests include software portability, software reusability, software engineering. and programming languages. He is a member of ACM, IEEE, AAAI, and Sigma Xi.

The Next Level of Reuse

by
Daniel E. Hocking, Computer Scientist

Army Institute for Research in Management Information,
Communications, and Computer Sciences
(AIRMICS)

Abstract: This paper summarizes the status and findings of an AIRMICS/Army/STARS research project to promote reuse by making it possible to produce, describe, validate, and find reusable components to use as pieces of a new system. The project addresses issues affecting reuse throughout the life cycle - both technical and administrative which currently limit the amount of reuse which can be realized. AIRMICS has organized a team of researchers from Martin Marietta and seven universities to investigate issues in reusability and in measurement of reusable components. This team will assess the effect of reuse on the validity and adequacy of tests of reusable components, develop classification techniques for reusable components to aid retrieval identify library issues affecting reusable components, extend database management systems for storing reusable components, identify management issues for designing components for reuse including incentives and legal issues, assess existing software components for reuse and automatic transformations for enhancing reuse, assess the effect of reuse on reliability as components are moved into new environments.

Background: This paper summarizes the status and findings of an AIRMICS/Army/STARS research project to promote reuse by making it possible to produce, describe, validate, and find reusable components to use as pieces of a new system. The project addresses issues affecting reuse throughout the life cycle - both technical and administrative which currently limit the amount of reuse which can be realized.

AIRMICS has organized a team of researchers from Martin Marietta; Georgia Institute of Technology; Georgia State University; Morehouse College; University of Colorado, Boulder; University of Houston, Clear Lake; University of Maryland, and University of Purdue to investigate issues in reusability and in measurement of reusable components. This team will assess the effect of reuse on the validity and adequacy of tests of reusable components, develop classification techniques for reusable components to aid retrieval, identify library issues affecting reusable components, extend database management systems for storing reusable components, identify management issues for designing components for reuse including incentives and legal issues, assess existing software components for reuse and automatic transformations for enhancing reuse, assess the effect of reuse on reliability as components are moved into new environments.

These issues are among many which need resolution to make reuse of functional components more common. This paper presents findings to date and the status of the efforts of the team members.

Levels of Reuse. The July 1987 issue of IEEE Software magazine focused on the theme "Making Reuse a Reality." Five of the seven articles in that theme are accounts of projects that have made the reuse of software components a reality.(1) Reusability as a concept is as old as computing. Initially coding was done in machine language. This was an intense manual effort to translate and cross-reference specific code for a specific machine. Later came assemblers that took an assembler language and converted that to machine language. The assemblers did much of the translation and in effect did much of the job of translating human readable information into a useful computer program. The assemblers were "reusing" concepts and taking a burden from the programmer's job and automating it. Later came "high level languages" and the compilers that converted the source code to object code. Much of the programmers job was transferred from thinking about the needs of the computer to thinking about the needs of the people who were to use the application. In most cases, the programmer was still putting together a procedure - but it was a procedure that was logical in people terms. Applications have begun to be reused as people learn to build systems that can be tailored within a company to the various locations where the company operates. More recently the industry has been dramatically changed by the availability of spread sheets and data base management systems for even the personal computer classes of machines. With each advance in the state of the tools, productivity of both programmers and people using the products of programmers has improved.

The next level of reuse is a change from what has gone before. To this point, reuse has been either at the level of primitives in the language being used, or algorithm reuse as in the math libraries, or whole systems as in the Information Systems Command where the Army has developed over a hundred standardized systems. The next step is to have libraries of components that can be used by systems and software engineers to develop new applications. These components can be simple components that do a specific task or they can be complex components that provide a framework into which more specific components can be inserted.

For management information systems which so far have been programmed mainly in COBOL, a major advantage of changing to Ada is the promise of reusable components. Another advantage is improved portability. Other advantages, such as moving to a single language, have long been achieved. COBOL, while not as portable as originally intended, is the single language of these systems. The modularization features of COBOL have high overheads and all data within a single component is global and available everywhere within the component.

Ada possesses several features that make reuse more feasible. The first of these is strong data typing. While programmers find this feature annoying at first, it has the advantage of restricting use of specific items of data to the intended use. This reduces errors involving the use of a wrong data item. Another feature is the separation of the specification and the body of an Ada component and the enforcement of data types across component boundaries. Since much of the data type checking is done at compile time instead of at operational time, this does not result in an unreasonable overhead. Furthermore compile time data type checking improves the ability to reuse components without introducing context errors for the data elements in the reused components. The concept of data hiding within a component is also a feature of Ada which promotes reuse of components. When data names are only available for use within a component, unintended use of identical data names causes no problem within the computer.

Components: I have made use of the term components without defining it. Within the context of the project, a component is an identifiable part of a system. It may be an Ada package or the design of the component which leads to an Ada package or the requirements which lead to the design. Alternatively, it may be the test plan for testing an Ada package or the documentation specified by the DoD Standard for software development, 2167A. This documentation may go beyond the specification part of an Ada package to describe what the Ada package is, how the Ada package works, and why the Ada package has been assembled. This purposely vague definition of a component is being used with the hope that the linkages among the forms of a component can be maintained and exploited whenever new components are being contemplated or worked on. It is this broad definition of components which we believe promotes more effective reuse of components from a resource point of view and which will make a life cycle emphasis on reuse a reasonable empnasis.

Life Cycle Emphasis for Reuse: Much of the effort in reuse to this point has been on the reuse of code or the reuse of some specific form of the software. This project focuses on the reuse of components at any point in the life cycle. The capability to trace the origin of the component from documents logically earlier in the life cycle such as requirements documents or the design documents is one of the goals identified in this project. The companion capability to trace the implementation of the component in one or many forms depending on the hardware environments or language environments in which the component has been realized is also one of the goals identified in this project. This project also focuses on the organizational aspect of reuse as compared to the personal aspect of reuse. We wish to make it possible for a project team to reuse components on a planned basis instead of leaving reuse to individual choices.

Underlying Assumptions: There are some basic assumptions that affect the choices being made in this project. These assumptions are:

Libraries will have hundreds and perhaps thousands of components to choose from when fully stocked. The logical outcome of this assumption is that reuse by inspection of the components will not be possible. This also means that the relationships among the components will not be obvious without automated support.

The history of the development, the support, and the use of the components will be essential to having a quality library that systems and software engineers can use with confidence. The logical result of this assumption is that a significant requirement for on-line storage of information about the components will exist. We have also made the assumption that on-line storage of this magnitude will be available for use in this way.

There is a set of descriptors that can be used for retrieval of components. The goal is to achieve a capability where there can exist "a Library of Congress" or a "corner store" for software components and that indeed any level of complexity between can exist.

Current operations will not scale up gracefully to handle our assumptions involving large numbers of components with large amounts of information about those components. History has shown that in the software area, solutions to simple problems have rarely scaled up easily to handle the same problem in a larger environment. In the contrary sense, however, software engineering history has frequently shown that simple problems are a special case of some larger problem. An example of this is the case of operating systems. An operating system established for a single cpu with a single user operating in a batch mode was difficult to upgrade to handle multiple users at the same time. Similarly, the multi-user operating system was difficult to upgrade to handle the case with multiple cpu's working together even when the cpu's were "tightly coupled." In each of these cases, however, the simpler case is just a special case of the largest case. The result of this assumption is that we need a design that handles components from all stages of the life cycle. The design must also handle links among these components and handle such complications as multiple copies of a single design in equivalent forms on different hardware, and alternative forms for a given set of hardware with slight variations in function. If such a complex case can be handled, we would then propose to show that the simpler cases are but special cases of the more complex case.

- Productivity improvements will result on a continuing basis from the capability to reuse software components. Many of the past breakthroughs resulted in step increases in programming productivity. Once that step increase was realized throughout the industry (as in the use of compilers), little further increase in productivity could be realized. The productivity improvements resulting from success in this and similar projects will be just as important as earlier breakthroughs. Once that initial step is taken and the initial productivity improvements achieved, additional, but smaller improvements in productivity will continue to be made as the library of reusable components expands.
- At least some of the programs already in functional libraries may be reusable. This means that evaluations of components and even some automated transformations may be appropriate to make such programs useful as components in new systems.

Goals The goal of this project is to develop a concept of operations that support large scale libraries of reusable components. The components in the library will be of varying scales of size and complexity and may be either single function components or large "application architectures" that can be completed by filling in the missing parts with either new components or by smaller already existing components.

The principle product of this project will be a handbook for reusing software. We will cover the development of new components that will be reusable, the transformation of existing components into more reusable components, the search and retrieval of components, the evaluation of components, and the tracking and record keeping of the use of components.

There will also be prototype demonstrations of some tools needed in such a complex undertaking.

Current Status: The University of Maryland is identifying:

- What structural and other product characteristics of an Ada component enhance its potential reuse value.
- How those characteristics can be assessed quantitatively.
- How the measurement process can be supported by automated Ada metric tools.
- What constructive guidelines can be derived from the above analysis.
- What guidelines for the reuse process there should be in Ada projects.
- How to support the transition process on Ada projects to emphasize reuse.

The principle investigators for the University of Maryland are Dr. Vic Basili and Dr. Dieter Rombach.

Purdue University is developing methods to use the operational history of reused software components to predict the fault-tolerant capabilities of new systems and the reliability of new systems that use those components. Keys to their approach include:

- The development of an iterative model that permits increasingly accurate predictions of fault-tolerance based on prior experience.
- Development of a definition of figures of merit that can be included in the interface specifications of Ada reused components.
- Development of aggregation rules that permit the composition of system-wide metrics from component-level figures of merit.

The principle investigator for Purdue University is Dr. Richard DeMillo.

Georgia Institute of Technology is identifying the effect of using reusable components on the testing process. Some of the issues they are dealing with include:

- Should components be developed for reuse or alternatively developed and reused?
- How does reuse affect the verification & validation of non-functional requirements?
- Can truly hardware independent software be developed?
- Can software be developed that will run correctly in any use context?

The principle investigator for the Georgia Institute of Technology is Mr. Mike McCracken.

Georgia State University is investigating library tools for use in describing, classifying, cataloging, organizing and managing Ada reusable components. The focus of their effort is on the development of classification definitions and methodologies, evaluation of these methodologies as applied to reusability generally and to Ada components specifically. The principle investigators for Georgia State University are Mr. Ross Gagliano and Dr. Scott Owen.

Morehouse College is investigating data management methodologies that promise merit for application to reusable Ada libraries. Various data management techniques are being investigated and evaluated for their effectiveness, practicality, functionality, and serviceability when used on Ada objects that are contained in a reusable Ada library. The principle investigator for Morehouse College is Dr. Arthur Jones.

University of Houston is developing a "Conceptual Model of the Software Engineering Life Cycle" that will promote reuse. This model must accommodate management of systems that evolve incrementally over a long life, contain nonstop components, and must simultaneously satisfy a prioritized balance of mission- and safety-critical requirements for runtime behavior. It must provide the capability to specify tools through the data objects that are created, changed, and used by those tools and the functions needed to manage those components. The concept is to enable the tracing of components from the first statement of requirements through multiple designs, implementations, and uses so that information from all stages in the development and use of a component can be used to determine the appropriateness of reuse of a component. Dr. Charles McKay is the principle investigator for the University of Houston.

The University of Colorado efforts are directed at using a richer semantic model than the entity-relationship model and at using an object-oriented database system. They are exploring combining these two approaches into one since together they provide both structural and behavioral ways of managing complex data objects. In earlier work the University of Houston has developed a prototype database management system with the capability of handling complex data objects. They are building on that prototype in this project and have identified constructs their prototype does not support. These include conglomerate objects for representing such things as software configurations, long and nested transactions for supporting interactive software design transactions, and more relaxed forms of concurrency control. They are identifying appropriate directions for solving these problems. Dr. King is lead investigator for the Universty of Colorado.

Martin Marietta Energy Systems has the responsibility of melding the results of the above efforts into a single reusability guide that can be used to direct further efforts in making reuse a reality. They will also be taking advantage of prior STARS work in producing a reusability guide and melding the results together. Pete Lesslie is the principle investigator for Martin Marietta.

Plans

There will be a formal IPR 15 and 16 June, 1988 involving presentations from each of the Universities and from Martin Marietta Energy Systems. Representatives of government agencies, universities, and private corporations will be invited to attend and comment on the progress to that point. Individuals wishing to participate in the IPR as a representative of their organization should send their name, address and phone number to:

AIRMICS (attn: Mr. Daniel E. Hocking)
115 O'Keefe Building
Atlanta, GA 30332-0800
Arpanet: RARI@ISEC-OA
Phone Number: (404) 894-3110

Presentations of the final work will be made at the Empirical Foundations of Information and Software Sciences Symposium, October 19-21, 1988 in Atlanta, Georgia. Questions about the symposium or offers to present other work on reusability should be directed to the above address.

Bibliography

1. Reusability Comes of Age by Will Tracz of IBM Federal Systems Division and Stanford University; pp 6-8 in IEEE Software, July 1987.

ADA: ALIVE AND WELL IN THE ARMY

Cira Mortgillo
Charles J. Walton

Grumman Aeorspace Corporation
U. S. Army, CECOM, Ft. Monmouth

Ada is alive and well in the Army. The Army Test Program Set Environment (ATSE) is one of the first software systems written primarily in Ada that has been delivered to the Army. ATSE is comprised of approximately 250,000 lines of Ada and is presently hosted on both the VAX series of computers under Ultrix, and the SUN workstation under UNIX.

The primary purpose of ATSE is to support the development and maintenance of ATLAS test programs. It is a structured environment consisting of a friendly, controlling User Interface, and a set of Test Program Set (TPS) tools to be used by a TPS engineer. These tools deposit configuration control information into a relational data base. A TPS consists of several items that are necessary to test a Unit Under Test (UUT) with appropriate Automatic Test Equipment (ATE). These items include a Test Program (TP), Test Program Instruction (TPI), Interface Device (ID's) and Supplementary Data (SD).

The components of the ATSE system parallel the Stoneman components of an Ada Program Support Environment (APSE). The Stoneman objectives include the followin: portability, open-endedness, and life cycle support. There are three levels to the Ada environment as recognized by the Stoneman requirement: the Minimal Ada Programming Support Environment (MAPSE), the Kernel APSE (KAPSE), and the APSE. The minimal set of tools is defined by the MAPSE. The KAPSE is the runtime library interface to the chosen machine's operating system. The KAPSE is all that need be changed if a different operating system is used. The APSE is a set of recommended tools that support the Ada system and include all other tools that can help in the overall system's life cycle.

Stoneman specifies the basic requirements of the APSE:

o Must provide an interface that allows an APSE user to interface with the APSE software as well as an interface between the different APSE software components.

o Must provide a minimal set of tools.

o Must contain a database to act as the central point of the environment.

The interface controls the database and also controls the use of the tools. The tool set consists of automated aids for program development, maintenance, quality assurance, and configuration management that are to be used throughout the life cycle of an application. The database is the central repository for information associated with each Ada application. This information is stored throughout the life cycle of the application.

ATSE has a similar set of goals: the ATSE environment was designed to be open-ended, portable and provide life cycle support. ATSE, like the APSE, has a User Interface, a tool set and a database.

Access to ATSE is facilitated and controlled by the User Interface. The user may request any of the tools by selecting from a series of menus. The User Interface checks user privileges before invoking the requested tool. The menus may be easily modified or expanded by editing text files. Each menu is broken into segments and "chained" together by NEXT and PREVIOUS commands, which allow for forward menu scrolling and backward scrolling, respectively.

The User Interface presents a single interface to the user, whether the user is on the SUN or the VAX. This interface buffers the user from UNIX commands by supplying an easy to use file manager command repertoire.

Security is enforced by this controlling interface in several ways. For one, a user may only invoke a tool that he has the capability to execute. Also, a user may only access files that are in his directory or subdirectories.

The User Interface menu mechanism allows for a very open-ended design: tools may be added to an existing menu and/or new menus may be added, without changing the User Interface program.

The ATSE tool set is an integrated tool set that is used to help build TPS programs. These tools include:

o ATLAS Compiler: The function of the ATLAS compiler is to accept test programs written in the IEEE Standart 716 C/ATLAS test language, process the program and generate interpretive code (IC) which can be executed on a suitable ATE.

e.g., Intermediate Forward Test Equipment(IFTE).

o ATE Simulator: The ATE Simulator allows the TPS engineer to execute all or part of an ATLAS program without actually interfacing to any ATE hardware. The Simulator allows the verification of an ATLAS program whether or not ATE assets are available.

o Syntax Directed Editor: This tool provides the TPS engineer with a convenient means of constructing syntactically correct ATLAS test program statements. Once the user chooses an ATLAS verb, the SDE generates a syntactically correct 'skeleton' and the user is prompted to add the necessary test parameters.

o Text Editor: The Text Editor provides host-computer independent and terminal independent editing and document processing capabilities. This tool may be used to create new files or modify existing ones.

o Hierarchical Integrated Test Simulator (HITS): HITS provides the user with an automated means of generating fault detection and fault isolation information for digital UUTs. This is done by simulating a circuit and then applying stimuli, either manually or automatically, and tabulating the results.

o Automatic Test Program Generator(ATPG) Post Processor: The ATPG Post processor accepts the output files of an ATPG system (i.e., HITS) and automatically generates an ATLAS language program and associated data files used for testing digital UUTs.

o AN/USM-465A (PSP-BASIC) To ATLAS Translator: This software tool provides a vehicle for the automatic translation of digital test programs from the PSP-BASIC test language (used by AN/USM-465A tester) into the ATLAS test language (used on IFTE Base Shop Test Station (BSTS) and Commercial Equivalent Equipment (CEE) test stations).

As each tool executes, it deposits information in a relational database. This information includes statistics on tool usage, such as which user invoked a particular tool or the number of times a tool was accessed. This data may be extracted in an interactive, query mode by authorized users for analysis and summation.

The ATSE Database Manager is a general purpose database manager and includes such features as: data definition language, data manipulation language, multi-user capability, report generator and recovery capability.

The portability issue on ATSE was resolved in a number of ways.

o Each tool was written in Ada. Ada was designed to provide a machine-independent environment which is re-hostable and re-targetable. The target machine can change without modifying existing programs.

o The operating system chosen was UNIX.

o Each tool interfaces with the operating system through the ATSE Rehostable Kernel System Library. This library is functionally very similar to the KAPSE. It contains calls that perform such functions as: scheduling programs to run in background mode; reading/writing to files; returning the host name to the calling program, etc. When ATSE was ported from the VAX to the SUN, there were no programming chages required to interface to the new operating system.

ATSE is a very open-ended system. As already discussed, the menus may easily be expanded to accommodate new tools. Additionally, users may be added to the ATSE User Table via a simple procedure. Each tool may be easily enhanced by virtue of the modular design that Ada imposes. It is his open-ended design that will facilitate all future growth on ATSE.

ATSE has the capability to provide life cycle support. Each tool deposits statistical information into a database. SQL-type queries may be formulated to retrieve pertinent information on this data. For example, the information retrieved can shed light on the current state of the project and how far along each user is in the development process.

One of the tables that the tools deposit into contains the following column headings (sample data is also shown):

TOOL ID NO	USER ID NO	TIME STAMP	UUT NO	SESSION TIME
12	123	08/09/87 10:42:07	uut01	1 HR 27 MIN
06	52	08/09/87 12:12:05	uut90	0 HR 12 MIN

The types of questions that can be generated from this table are:

- o For a particular UUT, what was the total time spent in development?
- o What were all the tools that a particular user invoked since January? (This question would show at what stage the user was involved in the development process.)
- o What tools consumed most of the users time?

ATSE DEVELOPMENT

The ATSE development process has primarily been a Pascal to Ada translation effort. Existing modules were translated to Ada. In addition, design and coding of most new modules proceeded in Pascal and were later translated to Ada. This approach had two distinct advantages:

- o The design and coding phase could begin in parallel with an Ada learning curve. This way, no time was lost in putting out a product.
- o If Ada performance provided to be unacceptable, the Pascal system would be available as a fallback.

The translation process was mostly a mechanical process. An inexpensive translator (Pastran) was utilized that executed on the IBM PC under PC DOS.

Pastran operates as follows. This program has three passes. In the first pass, Pastran determines the syntax or structure of the given program. If Pastran cannot make a sensible determination of the syntax of the program, it will give up after the first pass. The Pascal source code will then have to be massaged further.

In the second pass, Pastran analyzes the semantics or meaning of the Pascal program. Pastran reports errors whenever it cannot make sense of a syntactically correct program. Pastran may make guesses to determine semantic meaning which may lead to further semantic errors.

The third pass of Pastran actually generates Ada code by taking the meanings found in the second pass.

The steps taken for the translation process were as follows:

(1) The Pascal source file was checked for all features that Pastran could not handle perfectly. For example, if the module was very large, it would have to be divided into smaller pieces.

(2) The Pascal source files were downloaded to the IBM PC from the VAX.

(3) The source file was then run through Pastran.

(4) The output from Pastran was massaged. The types of corrections that were needed were:

- o Dummy procedures that were included for Pastran were removed.
- o Pascal identifiers that are Ada predefined types i.e. the identifier 'string', were changed.
- o Undeclared identifier errors usually implied that the appropriate code (or package) needed to be included with the source file.
- o Operations on string objects had to either use 'slicing' or the object had to be padded with blanks to the declared length.

(5) The original Pascal source was compared to the final Ada source to make sure that they were equivalent.

It was during the refinement process (Step 4) that the greatest Ada learning curve occurred. During this process, Ada features were exploited to enhance the maintainability of the source code.

For one, by virtue of converting to Ada, the Ada library structure had to be imposed. An Ada program consists of a library of one or more program units, which can be compiled separately. Each program unit (or package) has a specifier containing information that must be visible to other units, and a body that contains implementation details. This means that a program can be designed, written and tested as a set of independent software components.

Because of this library structure, time is saved when recompiling a large system if a change is made or someting is added. This structure also enforces modularity which helps keep the ATSE design open-ended and very maintainable.

Additionally, the Ada exception mechanism was introduced during the refinement phase of the Ada translation. This mechanism allows for defining of exceptional situations, signaling their occurrence and responding to them.

If an exception is found during program execution, normal execution ceases and control is passed to an exception handler. These include predefined exceptions as well as user defined exceptions. After execution of the exception handler the current program unit terminates and control is passed to the previous level of software.

The advantages of the exception mechanism for ATSE are increased readability because the exception handlers are in a textually distinct part of the program, and the elimination of the need to explicitly check for errors that could occur at any time.

Future work on ATSE will be designed and coded in Ada with the elimination of the translation step. An Ada design will ensure software that is reliable, efficient, and easy to modify. Additional Ada constructs (i.e., tasking, generic packages) will be exploited when necessary, which will ultimately provide for a very portable and maintainable ATSE system.

The performance of the Ada version of ATSE has proven to be comparable to the Pascal version. The ease with which the ATSE software system was rehosted from the SUN to the VAX and vice-versa, has demonstrated its portability. The one drawback is that Ada executables are about 15% larger than Pascal executables. This affects speed to some extent. In the future, this factor will have to be considered and compensated for.

An effort to design and code new ATSE tools in Ada will begin in the near future. It is anticipated that with the experience gained to date, efficient and cost effective software will be produced.

Development of an Ada Environment for Communications' Protocol Testing

K. Bachmann, M. Busch, H. Gerhards and R. Molle

SCS Technische Automation und Systeme GmbH
GB Sondertechnik, Oehleckerring 40, D-2000 Hamburg 62, FRG

As part of a study on transport protocols in special LANs a Protocol Test System (PTS) was implemented in Ada. This paper is a report on the project. The test system simulates data transmission from user A to user B via two TCP transport layer protocols on one single computer system. In order to simulate realistic data transmissions the main components of the test system were implemented as independent Ada tasks. In addition the transmission behaviour of every task is able to be changed dynamically and independently using runtime parameters. The structure and implementation of the test system and its components will be described in detail from the viewpoint of an Ada user. Many advantages, some disadvantages and even problems were observed during the intensive use of Ada in this type of work.

1. Introduction

An increasing effort in networking between computers of different manufacturers running different operating systems can be observed today. As a consequence there is a great need for standardized communication software, which guarantees safe connection handling and reliable data transmission within such heterogenous environments.

The OSI (Open Systems Interconnection) basic reference model divides data communication into seven functional layers (Figure 1). These layers are ordered hierarchically from the physical layer up to the application layer.

Layer 4 is called the transport layer and is responsible for transparent and reliable data transfer between two end systems within a computer network.

From the viewpoint of network performance the evaluation, implementation and parameterization of a transport protocol is extremely important.

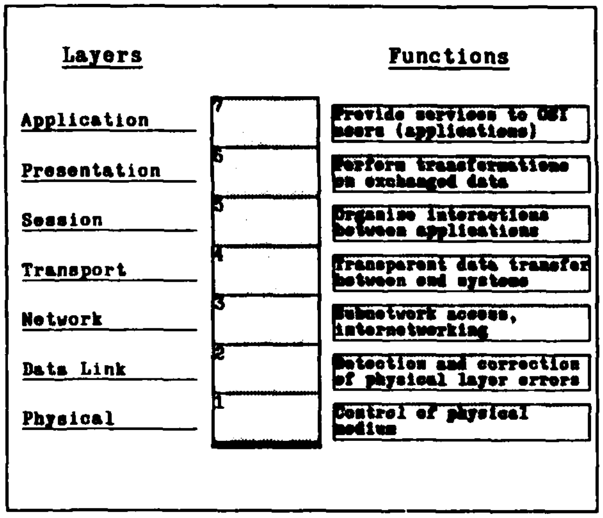

Layers
Functions
Application
Presentation
Session
Transport
Network
Data Link
Physical
7
6
5
4
3
2
1
Provide services to OSI users (applications)
Perform transformations on exchanged data
Organise interactions between applications
Transparent data transfer between end systems
Subnetwork access, internetworking
Detection and correction of physical layer errors
Control of physical medium

Figure 1: ISO Basic Reference Model for Open Systems Interconnection

Todays well known and most frequently used transport protocols for heterogenous computer networks are

- ISO-TP4: ISO Transport Protocol Class 4 and
- TCP : Transmission Control Protocol (MIL-STD-1778).

The specifications of these protocols allow software developers to vary the strategies for transmitting, retransmitting or receiving data and offer them a certain amount of freedom in the interface design to the ULP (Upper Layer Protocol). Besides these parameters the behaviour of a transport protocol depends on the properties of the network's lower layers 1 to 3 and the strategies implemented in its peer layer.

Since it is rather difficult to estimate dependencies of such parameters and their influence on data transmission behaviour, a Protocol Test System (PTS), which simulates TCP based data communication between two systems in one single computer, was implemented.

Figure 2 shows the functional structure of the PTS consisting of the modules:

- Simulation Control
- Transport Protocol Users on System A
- Transport Protocol on System A (TCP)
- Transport Protocol Users on System B
- Transport Protocol on System B (TCP)
- Network Simulation

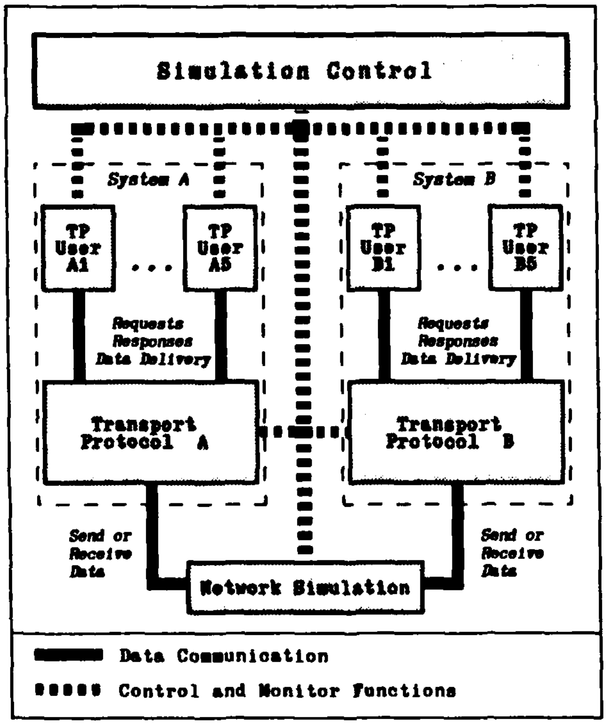


Figure 2: Functional Structure of the Test System

In order to build realistic scenarios the modules have to be implemented as independent processes which communicate and synchronize with methods of interprocess communication. In addition the central supervisor function for simulation control and monitoring shall have access to all system components at every time. For this reasons Ada with its tasking mechanisms seemed to be very suitable as an implementation language for the PTS.

In the following chapter a global description of the implementation of the PTS as well as some details, problems and their solutions are given.

2. Structure and Implementation

It is evident that the structure in Figure 2 is implemented by defining an appropriate Ada task type for each function. These tasks are administrated via the simulation control function. Unfortunately it is not possible to use the normal Ada task rendezvous mechanism to implement the control and communication pathways shown in Figure 2, because in an Ada rendezvous the calling task normally waits until the entry call is served by the called task. This mechanism is not suitable for the PTS, since the dependencies in the control flows of the various tasks would be too strong.

For this reason the PTS uses a more indirect and asynchronous way of intertask communication supported by server tasks called agents. Agents are independent tasks, which are created dynamically in order to accept data from one task, to deliver this data to another task and to exit after completion of data delivery. The originator task may continue with its mission immediately after a short delay caused by creation of the agent and the rendezvous with it. The agent then has to wait until the receiver task is ready to process the data destined for it. This mechanism for example allows a task to call its own entries, if required.

The agent mechanism offers the PTS several advantages:

- It is not necessary to implement an explicit administration for request queues.

- There is no direct data coupling between the communication tasks. It is therefore possible to implement some adaptation code within the agent task in order to integrate another transport protocol into the PTS (for example ISO-TP4).

A disadvantage encountered was the fact, that the memory space used by a task is not deallocated entirely after its completion unless the immediate master of the task is terminated. On the other hand control cannot leave an immediate master until all tasks created in its scope are terminated. To achieve complete independence of an agent a package has to be defined as its immediate master. That would cause a slowly increasing waste of memory space if no special housekeeping is done.

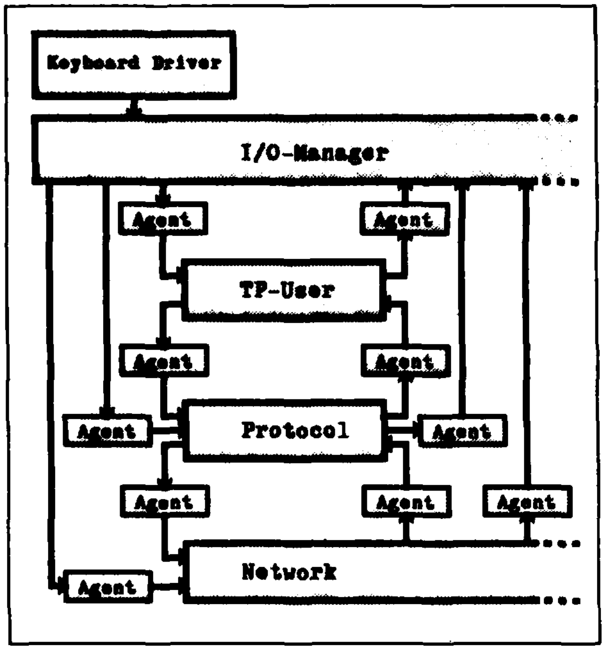


Figure 3: Task Structure of the Test System

Figure 3 shows the task structure of the PTS in detail. For simplicity only half of the system with only one TP-User is depicted. In order to allow user console interactions at every time, the simulation control module is implemented in two tasks, the Keyboard Driver and the I/O-Manager. The Keyboard Driver only waits for user console input and delivers these data to the I/O-Manager immediately. The I/O-Manager checks and executes console commands or respectively returns an error message. Additionally the I/O-Manager handles all PTS output activities (screen output, logfiles etc.).

Another important objective of the PTS proj'ct was to develop a portable implementation of the specific transport protocol (the TCP). Therefore a reasonable module structure was necessary, which separates hardware and operating system dependent code from the more general packages. Within PTS all functions have been implemented by using Ada features. Queue management is based upon dynamically linked lists. All interprocess communication is based upon the task rendezvous mechanism. When implementing the TCP on another system these mechanisms might not be adequate.

Ada private declarations have been used to encapsulate data types within some packages, such as timer administration, send/receive queue management and interfaces to the upper and lower layer protocols, since these packages may probably be modified during an exchange of the underlying system. The encapsulation of types will give greater safety against side effects after a redesign of these packages for other systems.

2.1 Special Data Types

This paragraph gives a few examples how some Ada features have been used to solve problems during the transport protocol implementation.

Sequence Numbers:
To identify and reorder protocol data units the TCP uses sequence numbers counting each byte in the data stream. Within the TCP a great deal of arithmetic and comparison operations are executed on sequence numbers. Since these operations are governed by special rules, sequence numbers have been implemented as a new data type and all necessary arithmetic and comparison operators have been overloaded by functions on this type.

Protocol data units:
Within the TCP header bit positions are defined to hold some special flags which are to be set or reset by the protocol. Pragma PACK has been used for the record types concerned in order to force a correct storage representation of the protocol

data units. This is very important since protocol data units appear as simple blocks of data to the underlying network layer.

2.2 A Model of the Transport Protocol Task

For this paragraph we regard the transport layer of the OSI reference model as a piece of software which can

- process events generated by its lower layer,
- process requests for its upper layer,
- generate service responses for its upper layer,
- generate events for its upper layer, and
- process timeout events initialized by its own protocol mechanisms.

Additionally the implementation of the transport protocol task was governed by the following requirements:

- The implementation of the protocol task shall only contain code for intertask communication and timeout control. All protocol specific code shall be implemented in separate packages which shall be used in the context of the protocol task.

- All static control information and runtime parameters used by the transport protocol (i.e. connection states, current buffer sizes, etc.) shall be visible to the simulation control task at every time.

- Multiple transport protocol tasks shall exist simultaneously each of them using their own control information and runtime parameters.

- The action procedures called in the context of the transport protocol task shall not use any static data because this code must be callable simultaneously from several protocol tasks each of them having their own local context.

The following example is a simplification of the code actually implemented for the test system and describes how the above listed objectives have been reached.
The example can be formally checked by a compiler although the package bodies of TP_TIMER and TP_PROCEDURES are not contained in the code. Nevertheless Ada allows to fix all interface conditions between the protocol and the outer world without any knowledge of the protocol's internal rules.

```
-- ***********************************************
-- SPECIFY DUMMY DATA STRUCTURES AS INTERFACE TO
-- LOWER AND UPPER LEVEL PROTOCOLS.

package TP_INTERFACE is

  -- DESCRIBE DATA RECEIVED FROM LOWER LAYER

  type FROM_LLP is (IMPLEMENTATION_DEPENDENT);

  -- DESCRIBE DATA RECEIVED FROM UPPER LAYER

  type FROM_ULP is (IMPLEMENTATION_DEPENDENT);

end TP_INTERFACE;


-- ***********************************************
-- SPECIFY TIMER SPECIFIC DATA STRUCTURES AND
-- FUNCTIONS

package TP_TIMER is

  -- SPECIFY TIMER UNITS

  type TIMER_UNIT is (IMPLEMENTATION_DEPENDENT);

  -- SPECIFY A DATA TYPE TO IDENTIFY DIFFERENT
  -- TIMERS

  type TIMER_ID is (IMPLEMENTATION_DEPENDENT);

  -- SPECIFY A TIMER SPECIFIC DATA STRUCTURE
  -- CONTAINING ALL INFORMATION FOR TIMEOUT
  -- CONTROL

  type TIMER_STRUCT is (IMPLEMENTATION_DEPENDENT);

  -- RETURN TRUE IF A TIMER IS ACTIVE

  function TIMER_SET (T: in TIMER_STRUCT)
    return BOOLEAN;

  -- RETURN DURATION UNTIL THE NEXT TIMER EXPIRES

  function UNTIL_NEXT_TIMEOUT
    (T: in TIMER_STRUCT) return DURATION;
  -- SET A TIMER THAT WILL EXPIRE AFTER THE GIVEN
  -- AMOUNT OF TIMER UNITS AND GENERATE A UNIQUE
  -- IDENTIFICATION FOR EACH TIMER

  procedure SET_TIMER
    (T: in out TIMER_STRUCT;
     V: in TIMER_UNIT;
     I: out TIMER_ID);

  -- CANCEL THE TIMER SET BY THE PREVIOUS
  -- PROCEDURE

  procedure CANCEL_TIMER
    (T: in out TIMER_STRUCT; I: in TIMER_ID);

end TP_TIMER;


-- ***********************************************
-- SPECIFY DATA STRUCTURES AND PROCEDURES TO BE
-- USED IN THE MAIN LOOP OF THE TRANSPORT PROTOCOL
-- TASK.

with TP_INTERFACE;
with TP_TIMER;

use  TP_INTERFACE;
use  TP_TIMER;

package TP_PROCEDURES is

   -- DESCRIBE DATA STRUCTURES USED BY THE TRANSPORT
   -- PROTOCOL INTERNALLY. SEPARATE TIMER DEPENDENT
   -- INFORMATION

   -- DESCRIBE TIMER INDEPENDENT CONTROL STRUCTURES

   type TIMER_INDEP is (IMPLEMENTATION_DEPENDENT);

   type TP_STATIC is
     record
       TIMIND: TIMER_INDEP;
       TIMDEP: TIMER_STRUCT;
     end record;

   -- PROCESS DATA RECEIVED FROM THE LOWER LAYER
   -- ACCORDING TO THE PROTOCOL RULES

   procedure PROCESS_LLP_DATA
     (D: in FROM_LLP; S: in out TP_STATIC);

   -- PROCESS DATA RECEIVED FROM THE UPPER LAYER
   -- ACCORDING TO THE PROTOCOL RULES

   procedure PROCESS_ULP_DATA
     (D: in FROM_ULP; S: in out TP_STATIC);

   -- PROCESS ALL NECESSARY ACTIONS AFTER A
   -- TIMER PREVIOUSLY SET BY THE TRANSPORT
   -- PROTOCOL HAS EXPIRED

   procedure PROCESS_TIMEOUT
     (S: in out TP_STATIC);

end TP_PROCEDURES;


-- ***********************************************

with TP_INTERFACE;
with TP_PROCEDURES;

use  TP_INTERFACE;
use  TP_PROCEDURES;

package TRANSPORT_PROTOCOL_TASK is

   -- USE THIS DATA TYPE FOR CONTROL FLOW WITHIN
   -- THE TASK BODY OF THE TRANSPORT PROTOCOL TASK

   type TP_TASK_STATUS is (STOP, RUN);

   -- DEFINE THE STRUCTURE OF THE ENVIRONMENT ON
   -- WHICH A PARTICULAR TRANSPORT PROTOCOL TASK
   -- OPERATES

   type TP_WORK_STRUCT is
     record
       STATUS : TP_TASK_STATUS := STOP;
       CONTROL: TP_STATIC;
     end record;

   type TP_ENVIRONMENT is access TP_WORK_STRUCT;

   -- SPECIFY TRANSPORT PROTOCOL TASK.

   task type TP_TASK_TYPE is

      -- CALL THIS ENTRY IMMEDIATELY AFTER TASK
      -- GENERATION TO PASS A POINTER TO
      -- THE SPECIFIC ENVIRONMENT ON WHICH THIS
      -- TASK SHALL OPERATE

      entry TP_START (ENV: in TP_ENVIRONMENT);
    -- CALL THIS ENTRY TO SIMULATE DATA DELIVERY
    -- FROM LOWER LAYER

    entry TP_RCV_FROM_LLP (D: in FROM_LLP);

    -- CALL THIS ENTRY TO SIMULATE DATA DELIVERY
    -- FROM UPPER LAYER

    entry TP_RCV_FROM_ULP (D: in FROM_ULP);

    -- CALL THIS ENTRY TO SHUT DOWN THE TRANSPORT
    -- PROTOCOL

    entry TP_STOP;

  end TP_TASK_TYPE;

end TRANSPORT_PROTOCOL_TASK;


-- ***********************************************

with TP_TIMER;
use  TP_TIMER;

package body TRANSPORT_PROTOCOL_TASK is

  task body TP_TASK_TYPE is

    E: TP_ENVIRONMENT;

  begin

    -- WAIT HERE TO RECEIVE A POINTER TO THE
    -- OPERATING ENVIRONMENT

    accept TP_START (ENV: in TP_ENVIRONMENT) do
      E        := ENV;
      E.STATUS := RUN;
    end TP_START;

    -- MAIN LOOP OF THE TRANSPORT PROTOCOL TASK

    while E.STATUS = RUN loop
      select

        -- SWITCH STATUS TO STOP TO EXIT THE TASK
        -- MAIN LOOP

        accept TP_STOP;
        E.STATUS := STOP;
      or
```

```
        -- ACCEPT AND PROCESS LLP DATA

        accept TP_RCV_FROM_LLP(D: in FROM_LLP) do
          PROCESS_LLP_DATA (D, E.CONTROL);
        end TP_RCV_FROM_LLP;
      or

        -- ACCEPT AND PROCESS ULP DATA

        accept TP_RCV_FROM_ULP(D: in FROM_ULP) do
          PROCESS_ULP_DATA (D, E.CONTROL);
        end TP_RCV_FROM_ULP;
      or

        -- GENERATE TIMEOUTS FOR THE TRANSPORT
        -- PROTOCOL

        when TIMER_SET (E.CONTROL.TIMDEP) =>
          delay
            UNTIL_NEXT_TIMEOUT (E.CONTROL.TIMDEP);
          PROCESS_TIMEOUT (E.CONTROL);
      end select;
    end loop;

  end TP_TASK_TYPE;

end TRANSPORT_PROTOCOL_TASK;


-- ***********************************************
-- A VERY SIMPLE EXAMPLE HOW TO ACTIVATE THE
-- TRANSPORT PROTOCOLS

with TRANSPORT_PROTOCOL_TASK;
use  TRANSPORT_PROTOCOL_TASK;

procedure START_TP is

  -- GENERATE TASKS AND THEIR SPECIFIC
  -- ENVIRONMENTS

  TP_1:  TP_TASK_TYPE;
  ENV_1: TP_ENVIRONMENT := new TP_WORK_STRUCT;
  TP_2:  TP_TASK_TYPE;
  ENV_2: TP_ENVIRONMENT := new TP_WORK_STRUCT;

begin

  -- INSERT ADDITIONAL CODE FOR SPECIAL ENVIRONMENT
  -- INITIALIZATION HERE

  null;
  -- PASS ENVIRONMENT TO THE TRANSPORT PROTOCOL
  -- TASKS

  TP_1.TP_START (ENV_1);
  TP_2.TP_START (ENV_2);

  -- BEGIN WITH CODE FOR COMMUNICATION SIMULATION
  -- HERE

  null;

end START_TP;
```

3. Observations and Results

The Development System:
The development system for the PTS was a MicroVAX II running MicroVMS 4.5 and VAX-Ada 1.3. Additional support was provided by a language sensitive editor and the source code oriented debugger of VMS.

There were no problems to become familiar with the development software, since the Ada system, which works as compiler, library manager and interface to the VMS linker, as well as editor and debugger fit exactly into the normal user environment.

The language sensitive editor supports familiarization with Ada by generating syntactically correct templates of language constructs. In addition it is possible to compile programs without leaving the editor. If errors occur during compilation a report will be given in a second window on the screen and the developer can walk through the code directed by the locations of syntax errors. Simple errors (for example a missing ';') will be corrected automatically after confirmation by the developer.

The debugger is source oriented and offers all standard features such as setting breakpoints or changing values of variables. Besides that Ada specific support is provided like monitoring of task states and changing task priorities. During the development of the PTS a normal debugging session was run using one terminal for debugger input and output and another as PTS user console.

The Ada runtime system is not embedded in the operating system, i.e. Ada tasks are not processes in the sense of the operating system. Compiled and linked Ada programs run in the context of the user's process instead. All Ada specific runtime activities such as task scheduling is done within the program using a runtime library.

During the development of the PTS no significant problems were observed using this Ada system.

Personnel and Effectivity:
At project begin the development team's Ada know how was more or less theoretical. Nevertheless there was good skill in software development and in using the general facilities of the operating system environment.

The time spent on familiarization with Ada and the language specific development tools was rather short. Including this time and the time for testing and documentation the team consisting of three persons spent six month on the development of the PTS.

The Ada Influence on Software Development:
Compared with other languages (Pascal, C, Fortran, Cobol, etc.) Ada compilers check the source code more thoroughly. This offers the user the advantage to detect many types of errors at compile time already, which otherwise might cause runtime errors difficult to fix. Some examples are:

- Ada performs consistency checking between separately compiled units. This feature greatly simplifies software integration. Debugging is more or less restricted to program logic and executable instructions, since all interface relations have been checked at compile time.

- The private type feature encapsulates data and avoids unintended side effects.

- Consequent usage of the 'in' and 'out' mode specifiers in subprogram and task entry declarations forces detection of a lot of errors at compile time already.

- Using records with discriminants forces these values to be write protected. Unintended modification of a record discriminant will be detected by the compiler.

Conclusion:

This paper has detailed the development of a Protocol Test System for OSI layer 4 protocols in Ada. After careful consideration of advantages and disadvantages this approach has proved, that Ada is a suitable implementation language for communications software. Especially when autonomous entities in a simulation environment are required the tasking features of Ada seem to be very advantageous.

4. Acknowledgements

This research was supported by the Bundesministerium der Verteidigung (BMVg) under contract no. T/R526/G0010/A2511. Views and conclusions contained in this paper are the authors' and should not be interpreted as the official opinion or policy of the BMVg or any other person or agency connected with it.

5. References

BAR82 Barnes, J.G.P.,
Programming in Ada,
Addison-Wesley, 1982

ISO84 *Information Processing Systems*
Open Systems Interconnection,
Basic Reference Model,
ISO IS 7498/1, October 1984

ISO86a *Information Processing Systems*
Open Systems Interconnection,
Transport Service Definition,
ISO IS 8072, 1986

ISO86b *Information Processing Systems*
Open Systems Interconnection,
Connection Oriented Transport Protocol Specification,
ISO IS 8073, 1986

MIL83a *Reference Manual for the Ada Programming Language*,
MIL-STD-1815A, 1983

MIL83b *Transmission Control Protocol*,
MIL-STD-1778, 1983

Klaus Bachmann

Born 1953, mathematician, since 1985 with SCS (defense business).

Activities: Communications, interfaces and operating system software on VAX and microprocessors.

Michael Busch

Born 1957, electrical engineer, since 1985 with SCS (defense business).

Activities: Circuit switching design, communications, working on minis and microprocessors.

Heinz Gerhards

Born 1952, aeronautical engineer, since 1982 with SCS (defense business).

Activities: Gas dynamics, avionics and communications since 1977, working on microcomputers, minis and mainframes.

Rüdiger Molle

Born 1941, mathematician, since 1979 with SCS (industrial and defense business).

Activities: Head of a department dealing with communications and data processing system techniques.

All authors' adress is that of company.

DESIGN FOR FAULT TOLERANCE AND PERFORMANCE IN A DoD-STD-2167 Ada PROJECT

WALTER SOBKIW and THOMAS L.C. CHEN
E-Systems, Inc., ECI Division at St. Petersburg, Florida

ABSTRACT

As computer hardware decreases in cost, it becomes increasingly available to software developers and computer users. With this decrease, computer-based system tools are finding new automation applications, which tend to replace manual tasks. Most of these automation tasks require timely arrival and accuracy of their product or output.

This has resulted in the development of new requirements related to data integrity and system availability. The solution to preserving data integrity and providing for high availability has been to develop fault tolerant computer-based systems. This paper defines a fault tolerant system design methodology within the framework of DoD-STD-2167 and the constraints of implementation in Ada.

INTRODUCTION

The formal system methodologies utilized in the development of many of today's medium-to-large systems tend to only address the functional system requirements. They are usually top-down oriented methodologies which rely on structured systems analysis as developed by Yourdon and or Hierarchical Input Processing Output (HIPO) analysis as pioneered by IBM. The problem with these methodologies is that they tend to ignore other aspects of the system equally important to accomplishing the system mission. Specifically performance and fault tolerance in the system are not addressed in a methodical manner. In addition, the impact of various issues related to fault tolerance are not related to the functional analysis or the performance analysis of the system.

In the structured system analysis methodology, performance is considered to be a minor issue. The advocates generally assume 10% of the units need to be redesigned in any project to support any unexpected computer performance deficiencies. DoD-STD-2167 defines a system development methodology which relates system development products to major program milestones, however, the standard tends to only focus on the functional definition of the system with little emphasis on performance analysis and no emphasis on fault tolerant analysis.

This point of view has been disputed by many practitioners and system users. There are several articles documenting practitioners and system users point of view. These articles suggest that major effort must be dedicated to fault tolerance and computer performance in the early phases of a design project, and that this analysis be refined as the system design baseline matures.

To a very large extent, the maturation of the design baseline is heavily dependent upon the fault tolerant analysis and computer performance analysis of the system.

In the past, design for fault tolerance and performance of computer-based systems has been successfully achieved without a widely accepted design methodology. Examples of such systems include Nuclear Power Plants, the U.S. Air Traffic Control System.

Experience on an Ada fault tolerant communications system at E-Systems as well as experience in the design of the new U.S. Air Traffic Control System suggests that the design for fault tolerance and performance can be systematically accomplished by a series of analysis. These analysis are aimed at a certain class of questions to produce a set of documented design alternatives as the functional analysis of the system is performed.

These documented design alternatives can be eliminated or refined by the restrictions provided by the computer performance analysis, fault tolerant analysis, design restrictions of the hardware and operating system characteristics, and the functional analysis of the system. These studies and trade-offs must occur in a timely manner in conjunction with the functional design of the system to form the overall system architecture. To some extent the level of detail at each phase is driven by the 2167 standard.

The products associated with the design of a fault tolerant system include an availability model, a computer performance analysis, and a failure analysis. These products each take on a different form and address specific issues at the 2167 major milestones. Many of the fault tolerant key issues will only surface after a methodical detailed analysis of the system has occurred. The specific implementation of Ada is one of the key issues. The required solutions for many projects is several orders of magnitude larger than the solution depicted by the implementation independent function analysis of the system.

With the introduction of Ada, certain constraints are introduced into the fault tolerant design of the system. These constraints are generic in nature, related to the language and implementation-specific related to each vendors particular interpretation of the Ada specification. For example, the use of rendezvous as opposed to semaphore is a constraint on the design generic to the language definition, while the particular design characteristics of tasking for each vendor type is dependent upon that vendor's Ada implementation

PRESENT SYSTEM DESIGN PROCESS OVERVIEW

The present system design process is an interactive process in which certain key issues of the system design are surfaced as the level of detail of the design evolves. The design at the system level can be summarized as follows:

- Requirements Analysis
- Functional Analysis
- Functional Allocation

It is at this time, during system architecture development, that the fault tolerant strategy must be decided. For example, will the fault tolerant strategy be based on hardware, software, or both? Will the hardware and software in the system be built from the ground up or will the system be built using a combination of off-the-shelf hardware and software and newly developed hardware and software?

For example, Stratus Computer Systems [3], [7] fault tolerant strategy is based on hardware in which all the hardware is replicated in a dual-dual configuration. This fault tolerant strategy may make sense if the driving factor is fault tolerance in hardware as opposed to software. If there are issues associated with reconfiguration and recovery time, then August systems [6], [8] MR system may be of interest since its reconfiguration and recovery is inherent in its normal mode of voting operations. If fault tolerant software is a primary consideration, then perhaps Tolerant Systems [3] or Tandem Computers [3], [5] with their transaction processing facilities may be more appropriate to the application.

These two vendors also may be attractive if the fault tolerant software may use innovative software error detection mechanisms such as multiple designs processing the same data and comparing the outputs. [10]

If there is a tightly coupled application that requires extensive amounts of computer resources, then perhaps a fault tolerant system must be built from the ground up using IBM mainframe like computers. [1], [4], [8], [9], [11]

The IBM mainframe computers offer extensive hardware error checking capabilities with the CPU logic and multiple instruction engines that can be tightly coupled to shared memory.

As the system design process progresses to lower levels of detail, certain issues related to the use of Ada must be resolved. The first question is, will Ada be appropriate for the application in question? Additional questions would be: Will the entire system be Ada-based or will there be some subsystems implemented in C, for example? Will the same Ada-based computer subsystem share another environment such as Unix and will the Unix environment support another language such as Fortran or assembly? Fault tolerant implementation requires control structures. Will these structures be developed by the applications programmers in Ada or will co-existing operating system services provide these control structures?

These issues are surfaced during the course of designing fault tolerant Ada-based systems. However, the framework of 2167 makes it difficult to document the analysis and results that lead to final requirements definitions that resolve these issues in each particular vendor design.

FAULT TOLERANT COMPUTER DESIGN METHODOLOGY

Figure 1 identifies the major milestones in the 2167 system development methodology. Normally when the standard is reviewed and the major milestones identified, the design and management staff view the design from the main mission function which is to define, develop, and design the automation applications functions. For example, if an air traffic control system is being developed, then the primary focus is identifying what the system is suppose to do (i.e., define the system functions) and how the system will support those mission function (using micros, mainframes, array processors, etc.)

However the analysis of the system in terms of availability and fault tolerances becomes of secondary concern. More important is that the functions associated with a fault tolerant mechanism are ignored. Figure 1 also shows the major milestones in a 2167 program and the kind of activities associated with each major milestone. Missing are the products which show the fault tolerant design baseline at each major milestone and how the design effort arrived at the fault tolerant baseline design.

Figure 2 has identified the DoD-STD-2167 major milestones and mapped a family of recommended activities and products to support the design of a fault tolerant system. These products are summarized as follows:

- Fault Tolerant Design Program Plan
- Generic Fault Tolerant Analysis
- Availability Model
- Static and Dynamic Computer Performance Models
- System, Hardware, Software, and Operational Failure Analysis

	SRR	SDR	SSR	PDR	CDR	IMPLEMENTATION
PRODUCTS	- PRELIMINARY SYSTEM SEGMENT SPECIFICATION	- SYSTEM SEGMENT SPECIFICATION - PRELIMINARY SRS AND IRS	- SRS AND IRS - PRELIMINARY TLDD	- TLDD - PRELIMINARY SDDD	- SDDD - PRELIMINARY IMPLEMENTATION	- RELEASED VERSION
ACTIVITIES	- REVIEW AND MODIFY CUSTOMER A-LEVEL SPECIFICATION - SUPPORT THE DESIGN CONCEPT WITH ANALYSIS	- IDENTIFY SYSTEM FUNCTIONS AND INTERFACES - ALLOCATE SYSTEM FUNCTIONS TO GENERIC HARDWARE SUBSYSTEMS - SUPPORT THE DESIGN CONCEPT WITH ANALYSIS	- IDENTIFY CSCI - ALLOCATE SYSTEM FUNCTIONS TO CSCI - ALLOCATE CSCI TO GENERIC HARDWARE - SELECT LANGUAGE AND SYSTEM CONTROL ENVIRONMENT - SUPPORT THE DESIGN CONCEPT WITH ANALYSIS	- IDENTIFY ALL CSCI AND TLCSC - IDENTIFY ALL COTS HARDWARE AND SOFTWARE - ALLOCATE CSC TO HWCI, MAJOR COMPONENTS AND SUBSYSTEMS	- IDENTIFY ALL LLCSC AND UNITS - ALLOCATE LLCSC TO HWCI, MAJOR COMPONENTS AND SUBSYSTEMS - ALLOCATE UNITS TO ADA PACKAGES, PROCEDURES, AND TASKS - SUPPORT THE DESIGN CONCEPT WITH ANALYSIS	**CODE** - CODE - DATABASE POPULATE **DT&E** - TEST CONCEPT - TEST DEFINITION - TEST PROCEDURES - TEST RESULTS

Figure 1. DoD-STD-2167 Major Milestones and Expected Products

SRR - Fault Tolerant Design Program Plan

The System Requirement Review (SRR) is primarily used to review the customer specification. Normally, the system developer has an initial concept of the system and is able to identify problems associated with the customer specification. These problems include functional, computer processing performance requirements, and problems associated with availability and maintainability requirements. At this time, the primary focus of the effort associated with fault tolerant definition should be an examination of the mission and the availability requirements.

Many times the mission can be redefined with less stringent availability requirements and the system can exceed fault tolerant expectations. For example, in the case of the communications systems being developed by E-Systems, the availability requirements were defined with centralized communications queues in mind. Some of these systems can be made more fault tolerant with a distributed queue however, the customer specifications would have made those designs non-compliant with the overall system specifications. By SRR, the program plans should be developed and available to all personnel. This includes a stand-alone

	SRR	SDR	SSR	PDR	CDR	IMPLEMENTATION
PRODUCTS	- MODIFIED A-LEVEL SPECIFICATION REQUIREMENTS - FAULT TOLERANT DESIGN PROGRAM PLAN	- AVAILABILITY MODEL RESULTS - STATIC COMPUTER MODEL RESULTS - GENERIC FAULT TOLERANT ANALYSIS (COOK BOOK)	- REFINED AVAILABILITY MODEL RESULTS - REFINED STATIC COMPUTER MODEL RESULTS - FAILURE ANALYSIS. SEGMENT SPEC AND SRS REQUIREMENTS	- REFINED MODEL RESULTS - DYNAMIC MODEL RESULTS - REFINED FAILURE ANALYSIS SRS AND TLDD REQUIREMENTS	- REFINED MODEL RESULTS - REFINED FAILURE ANALYSIS TLDD AND SDDD REQUIREMENTS	- FAILURE ANALYSIS - SOFTWARE AVAILABILITY GROWTH PROGRAM PRODUCTS PACKAGE
ACTIVITIES	- REVIEW CUSTOMER A-LEVEL SPECIFICATION RELIABILITY AVAILABILITY REQUIREMENTS	- INITIATE AVAILABILITY MODEL - INITIATE STATIC COMPUTER PERFORMANCE MODEL - IDENTIFY GENERIC FAULTS AND RESULTING ERRORS - IDENTIFY GENERIC FAULT TOLERANT FEATURES	- REFINE AVAILABILITY MODEL - REFINE STATIC MODEL - IDENTIFY SYSTEM, SUBSYSTEM, AND SOFTWARE FUNCTION FAILURE MODES	- REFINE MODELS - IDENTIFY CRITICAL CSCI, TLCSC, AND HWCI. IDENTIFY FAILURE MODES - IDENTIFY CRITICAL OPERATIONS AND FAILURE MODES - IDENTIFY GENERIC ADA CONSTRAINTS AND SOLUTIONS	- REFINE MODELS - IDENTIFY CRITICAL CSCI, TLCSC, LLCSC, UNITS, AND HWCI IDENTIFY FAILURE MODES - REFINE OPERATIONS FAILURE ANALYSIS - IDENTIFY ADA IMPLEMENTATION SPECIFIC CONSTRAINTS AND SOLUTIONS	- FAULT TOLERANT PROGRAM PLAN - GENERIC FAULT TOLERANT ANALYSIS - SYSTEM FAILURE ANALYSIS - SOFTWARE FAILURE ANALYSIS - HARDWARE FAILURE ANALYSIS - OPERATIONS FAILURE ANALYSIS

Figure 2. DoD-STD-2167 Major Milestones and Recommended Fault Tolerance Analysis Products

plan to support fault tolerant design. The Fault Tolerant Design Program Plan identifies the inputs to the effort, the product outputs, and the relation of the fault tolerant products to all other products on the program.

SDR - Generic Fault Tolerant Analysis, Preliminary Availability, and Static Computer Performance Models

By Software Design Review (SDR) there should be a generic architecture solution. This solution identifies all system functions, allocates those system functions to generic data processing hardware, identifies the internal and external interfaces and provides supporting data to satisfy the system performance requirements. At this time, there also should be:
(1) supporting analysis in the tradeoffs that show the generic architecture solution; (2) the performance analysis identifying the computer processing performance at the function level; and (3) an availability model.

This is a very critical time period for the definition of the fault tolerant system, since effectively the basic approach to fault tolerance is chosen. This definition should include, for example, if the system is to be characterized with large numbers of hardware error checkers capable of detecting transient errors or is the system characterized with software capable of detecting errors in processing induced by the hardware or software. This time period has effectively identified the generic approaches to implementing a fault tolerant solution, developed computer performance and availability models, and traded off the various solutions in overall program goals using the inputs from the various models on each fault tolerant architecture approach.

During this time, the program should examine the state-of-the-art in fault tolerance. This should be in the form of a Generic Fault Tolerant Analysis document that identifies various fault tolerant features and their capabilities in terms of dealing with various hardware and software faults and errors. This document also establishes the fault tolerant definitions that will be used throughout the program development effort. These definitions will include various generic ways in which computer-based systems have failed. In addition, if there is some prior generation of the computer-based system in operation, this document identifies some of the more unusual problems encountered in that previous generation computer-based system.

This document needs to be detailed and yet capable of being reviewed and understood in a very short time period. This document effectively forms a 'cookbook' of generic fault tolerant approaches available to the system designers. Just as an A-Spec is developed to identify the mission of the system, this document is developed to identify various fault tolerant functions and their missions.

For example, a Cyclic Redundancy Check (CRC) attached by the software and maintained during all data transport processing can protect mission data to a very high degree from a host of various system faults. That knowledge may not be readily known or understood by all the system designers (nor should it be). In addition, each designer will have an opinion about the effectiveness of each fault tolerant function.

This document will, for example, provide the official program position on the effectiveness of range checking in software versus the use of hardware error checkers in the CPU hardware, even if the program position is based on qualitative analysis instead of quantitative analysis. The issue is to capture the analysis and make the findings available to all designers during system design.

SSR - System Failure Analysis, Refined Models

The SSR is primarily focused on software design. Effectively, the generic architecture defined for SDR has evolved to a more application specific system definition. The functions previously identified are now allocated to Computer Software Configuration Items (CSCI's). Those CSCI's are allocated to the generic hardware architecture configuration. At this time, the language should have been selected along with the system control environment. Will the system be implemented in Ada using a real-time Unix executive, or will the system be partially implemented in Ada and C with a custom designed real-time control executive? Certain specific implementation issues associated with these software design selections need to be identified and surfaced. In some cases, new functions need to be defined to support the top-level software architecture selection issues. At this time, the fault tolerant design of the system needs to continue refining the computer performance and availability models.

In addition, a formal failure analysis needs to be initiated. That failure analysis needs to begin identifying various failure modes associated with all system mission functions, defining the potential harm of each failure mode (some failure modes may be harmless), and modifying the system baseline from SDR to minimize the potential harm. The output of this failure analysis should be requirements that are incorporated into the system SRS's and the System Segment Specifications.

This failure analysis should be partitioned into system, hardware, software, and operations. The failure analysis should use the design products developed to date and primarily focus in on the SDR baseline design. The most important products are the identification of subsystems, major components, software functions in the form of data flow diagrams, and any single thread diagrams.

The failure analysis should begin by effectively performing a Black box analysis on the automation system where subsystems are removed from operational service because of a failure. Failed subsystems that result in damage to on-line operations and impact the availability requirements are the most critical subsystems. When a subsystem is removed from service, the impact of that event should be documented. Those impacts include loss of system data and mission functions. If critical data or critical mission functions are lost, then various solutions should be identified for preventing the loss of that data or system function service. A starting point for all fault tolerant solutions should be with the generic fault tolerant features identified in the fault tolerant analysis effort used to support the SDR baseline. Those generic solutions should then be modified to support the specific characteristics of the baseline.

Those approaches should be documented and a tradeoff in terms of the effectiveness of each solution and its impact on the system architecture identified. Many times the various fault tolerant solutions have large impacts on the computer performance characteristics of the system, and so the timing and sizing analysis needs to be closely coupled to this tradeoff analysis.

The primary focus at this time should be the system failure analysis, however, the hardware, software, and operational failure analysis can also begin at this time. The other failure analysis efforts will be refined during PDR.

PDR - Full Failure Analysis, Refined Availability Model(s) and Dynamic Computer Performance Model(s)

The Preliminary Design Review (PDR) consists of an architecture where all the interfaces are defined and allocated to a system configuration baseline, all the CSCI's, and Top-Level Computer Software Components (TLCSC's) are identified and their requirements are documented in the Top-Level Design Document. At this time, the computer performance model should have evolved to represent the design in terms of the TLCSC's.

In addition, a preliminary dynamic model representing various queue relationships and context switching baselines should be providing the designers with further definition of the system performance characteristics. The availability model should be tracking the evolving baseline. The architecture solution should be defined in terms of a vendor implementation if off-the-shelf hardware and or software will be used in the system.

The fault tolerant analysis should now switch to a failure analysis of the each individual TLCSC and Hardware Configuration Item (HWCI) in the system. The TLCSC's should be characterized in terms of importance in the system and the various failure modes previously identified at SDR should be allocated to the TLCSC's identified. In addition, each HWCI should be characterized in terms of importance in the system, and the various subsystem failure modes previously identified at SDR should be allocated to the identified critical HWCI's.

It is at PDR that the architecture solution has transitioned from a generic solution to a non-generic solution. The Ada language has been selected along with any Commercial Off-the-Shelf (COTS) hardware and software. In addition, the design of new hardware and software has begun to be constrained by various implementation issues.

For example, if a high speed communication bus is being developed the access mechanism should be defined. Is the bus Carrier Sense Multiple Access (CSMA) or Token controlled? Is the physical plant fiber optic or copper based? The same non-generic issues related to fault tolerance also should be surfacing at this time. Which TLCSC's and HWCI's are critical? For the critical TLCSC's and HWCI's, what are the error detection mechanisms? For the critical TLCSC's and HWCI's, what are the recovery mechanisms? How long does it take to recover and will the availability be satisfied? These are all very detailed questions that cannot be reasonably answered without a non-generic architecture definition.

It is at PDR that the generic constraints of Ada begin to surface. Fault tolerance is accomplished with error detection, replication of the same design or alternate designs, and routing via active healthy paths. In the non-generic architecture, these functional requirements translate to control structures.

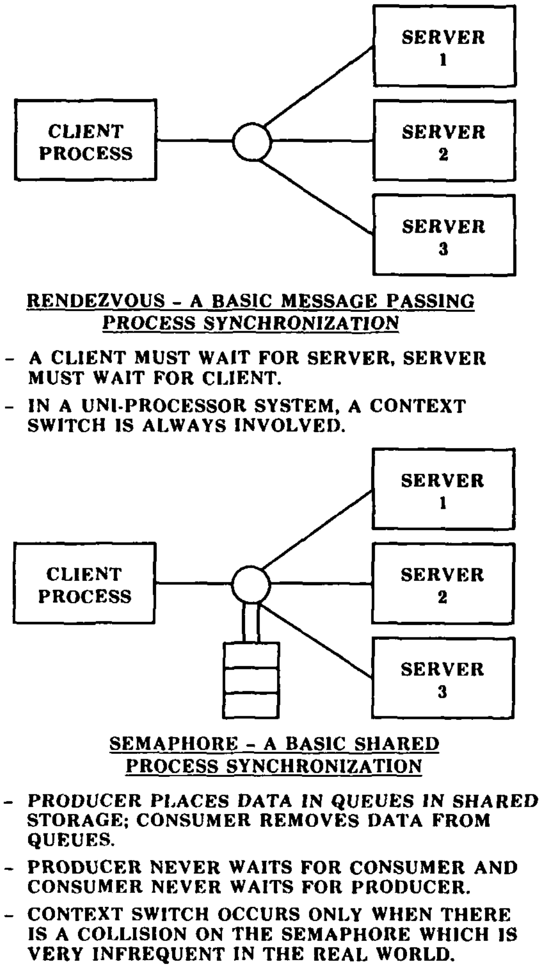
For example, to protect data from damage resulting from memory storage failure data is replicated. This replication can occur in hardware or software. However, after examining the market place there are no vendors that offer full data replication implemented in hardware. This control structure must either be implemented in the applications code or the operating system services.

In addition, to maintain consistency before, during, and after a failure, the mechanism grows into a sophisticated sequence or control structures filled with many potential design errors in and of itself. In addition, once an error is detected by either hardware or software, recovery must proceed. This recovery also must be in a consistent manner to ensure that system data is not compromised. These are all elaborate control structures that must be implemented in Ada by the applications programmer from the ground up.

This issue is further exacerbated by the context switch time of many Ada implementations. In order to implement many of these control structures in Ada context switching needs to occur. This context switch time can significantly reduce the time left to support application programming functions. This issue is particularly acute in real-time fault tolerant system design.

Some vendors such as Tolerant systems have developed control structures and made them available to the applications programmer. In Tolerant's case, these control structures reside in an augmented Unix look alike operating system. The Ada 'application' code executes with the augmented Unix look alike operating system.

It is at this time that the issues related to interprocess communications and process synchronization surfaces. There are two basic process synchronization schemes as depicted in Figure 3.



RENDEZVOUS - A BASIC MESSAGE PASSING PROCESS SYNCHRONIZATION

- A CLIENT MUST WAIT FOR SERVER, SERVER MUST WAIT FOR CLIENT.
- IN A UNI-PROCESSOR SYSTEM, A CONTEXT SWITCH IS ALWAYS INVOLVED.

SEMAPHORE - A BASIC SHARED PROCESS SYNCHRONIZATION

- PRODUCER PLACES DATA IN QUEUES IN SHARED STORAGE; CONSUMER REMOVES DATA FROM QUEUES.
- PRODUCER NEVER WAITS FOR CONSUMER AND CONSUMER NEVER WAITS FOR PRODUCER.
- CONTEXT SWITCH OCCURS ONLY WHEN THERE IS A COLLISION ON THE SEMAPHORE WHICH IS VERY INFREQUENT IN THE REAL WORLD.

Figure 3. Rendezvous vs True Semaphore

The first one is based on a semaphore and requires shared memory. When this scheme is selected the producer process deposits its products in the shared memory and the consumer process takes the products off the shared memory. The semaphore is used to arbitrate the access of the shared storage. A process context switch is only required when there is a hit on the semaphore. This is an old efficient scheme which requires the support of shared memory. The second scheme is message passing. Two processes are synchronized by expressly passing messages to each other. When both processes share the same hardware processor, a context switch is always required in synchronization. When each process is supported by a different hardware processor, one of the processes will be idle waiting for the other process unless the duty cycle of both processes are perfectly matched. Tasking in Ada is synchronized by rendezvous, which is a message passing scheme.

In the semaphore scheme, the application context and the queues need to be secured for fault tolerance. In the message passing scheme, the processes (task table, stack, etc.), as well as the application context, has to be secured for fault tolerance.

A project can chose to use Ada tasking or an operating system process in place of Ada tasking. The choices of Ada will also effect the selection of the hardware. No Ada compiler in existence today can allocate Ada tasks to a different hardware processor nor is there any support for rendezvous across hardware processors. The nature of the application may favor one scheme or the other. The hardware architecture should match the selected scheme. The architecture design must weigh all these interdependent matters to form the lowest cost solution that is technically practical. Many designers would argue that this is an implementation issue, however, these two choices can severely impact the architecture to the point of invalidating the architecture concept.

CDR - Refined Products ready for delivery to Software Availability Growth Organization

By CDR, the requirements, analysis, and design documentation should be complete. This milestone effectively gives the go-ahead for the contractor to begin implementing all the hardware and software. It is at this time that the Ada packaging and procedure concepts should be allocated to the TLCSC's, LLCSC's, and units.

At this time, the failure analysis has focused on the units, HWCI's, and resolved all issues related to the system databases. The issues related to Ada packages and procedures also have been resolved with a clear indication of how each package and procedure will be supported with fault tolerant features. The error detection, reconfiguration, and recovery mechanisms should be fully documented as requirements in the specifications and these requirements should be justified in the availability model and analysis, dynamic model and analysis, and the failure analysis.

The failure analysis should have identified all the design constraints associated with developing a real-time fault tolerant Ada system using the selected hardware and accompanying software. This analysis will then be submitted to the organization tasked with the software availability growth program. This software availability growth program should be fully established and ready to begin its activities with the test and integration phase during implementation.

It is at this time that the vendor implementation of Ada will begin to constrain the design of the system. For example, the Ada task is not constrained in DoD-STD-1815 and there are a number of different vendor implementations of this Ada task some of which do not support true parallel or concurrent processing. This issue is particularly acute in a communications system where concurrent I/O is to be supported. There are also issues related to the Ada load image and the management of the heap. In both cases, memory is consumed such that embedded real-time fault tolerant systems can be severely constrained. These issues need to be surfaced and resolved by CDR.

CONCLUSIONS

In conclusion, the design of a fault tolerant system is similar to the design of the main functional application of that real-time system. The fault tolerance analysis needs to begin with a plan. That plan should not only contain the approach and methodology to supporting the fault tolerant design, but it should also contain a heavy emphasis on the technical aspects of fault tolerance. Once this plan and cook book are established, then a failure analysis of the system needs to occur at each level of the system design. This begins with the subsystems and top-level functions and concludes at the Ada packages or procedures and the HWCI's. The entire failure analysis should then be packaged as one product and submitted to the software availability growth program organization.

During the course of the design, many control structures used to support fault tolerance will need to be developed. Some of these control structures have been implemented in external operating system environments. Other structures will need to be implemented in Ada and will be limited by context switching time. In some cases, some of the fault tolerant control structures may need to be developed in assembly language to preclude the negative impacts of context switching time.

As with all design methodologies, the intent is to provide a vehicle for identifying and controlling program risk. Risk in this case is related to the successful design of a fault tolerant system. With the identification of these products and related activities schedule, cost, resources, and expertise can be identified and planned to support a successful fault tolerant design. More importantly, the progress of the fault tolerant design activity can be effectively tracked during the entire design process.

REFERENCES

1. IBM Journal of Research and Development, IBM 3081 System Development Technology Vol 26, Number 1, Jan 1982.

2. Computer System Isolates Faults - The Tolerant Systems Eternity Series, Computer Design, Nov 1983.

3. Fault Tolerant Systems in Commercial Applications, Omri Serlin. IEEE Transactions on Computers, Aug 1984, pp 18-30.

4. Fault Tolerant Computing - Concepts and Examples, David A. Rennels,Vol C-33, No. 12, pp 1116-1129. IEEE Transactions on Computers, Aug 1984.

5. Fault Tolerant Architectures Douglas Eidsmore, Digital Design, pp 70-82, Aug 1983.

6. Fault Tolerant Computers Ensure Reliable Industrial Controls Electronic Design, 25 Jun 1981.

7. Making Processing Fail-Safe Robert Fredburg, pp 255-264. Mini-Micro Systems, May 1982.

8. Fault Tolerant Computer Study, Jet Propulsion Lab, JPL Pub 80-73, Contract NAS 7-100, pp 2-1 to 2-51.

9. Survey of Fault Tolerant Computer Security andComputer Safety SRI International, NTIS RADC.TR-86-164 I-1 to I-26, IV-1 to IV-56.

10. An Empirical Study of Software Error Detection Using Self-Checks. Fault Tolerant Computing Symposium July 1987. Sung D Cha, John C. Knight, Nancy G. Levenson, and Timothy J. Shimeall, pp 156-162.

11. Fault Tolerance Principals and Practice, T. Anderson and P.A. Lee Computing Laboratory, University of Newcastle upon Tyne, England. I-1 to III-89.

Biography

Walter Sobkiw is a Senior Principal Engineer with E-Systems, ECI Division. He is directly responsible for failure analysis, simulation testing, and automation systems for military communications systems. He holds a BSEE from Drexel University.

Thomas L.C. Chen is a Member of the Technical Staff in the Software Systems Department, E-Systems, ECI Division. He is the Principle Software Designer of Survivable Communications Systems, has over 20 years experience in the development of communications methodology. He holds an M.E. from Taipei Institute of Technology.

The Development and Retargeting of SECOMO in Ada

John LeBaron   Raymond Menell
Judith Richardson   Johnny Ng

U. S. Army Communications Electronics Command
Center for Software Engineering
Fort Monmouth, New Jersey

Abstract

This paper discusses the experiences gained and lessons learned during a two phased software development project using Ada. The first phase focused on the conversion of the Software Engineering Cost Model (SECOMO) from VAX/VMS FORTRAN to VAX/VMS Ada. The second phase concentrated on retargeting the Ada implementation from VAX/VMS to IBM PC/AT and XT machines running under MS-DOS.

The work described in this paper was conducted at the U.S. Army's CECOM, Center for Software Engineering at Fort Monmouth, N.J., during the period January 1986 to May 1987.

Introduction

The Ada Language Branch is part of CECOM's Center for Software Engineering at Fort Monmouth, New Jersey. Our mission is to support the Center's Project Leaders and CECOM's Program Management Offices, by increasing their awareness of Ada issues, promoting AMC's policies and helping them to insert Ada-based software engineering requirements into their application domains. Our assign task was to select an existing system and implement it in Ada. The project had the following goals:

- To develop a usable product,
- To provide in-house Ada programming experience,
- To use Ada-based software engineering methods and tools during software development.

We evaluated several candidate systems for this project. Our evaluation criteria included amount of source code that would allow the project to be completed within six months to a year, availability of a target environment, documentation, and some experience in the system's original implementation language. The selected system was the Software Engineering Cost Model (SECOMO) developed by IIT Research Institute, Rome, N.Y. This system consisted of approximately 10,000 lines of FORTRAN source code.

The SECOMO program is an interactive software cost estimation model that calculates the total staffing requirements of a software system and includes activities related to a Life Cycle Software Engineering Center. The SECOMO program is based on the intermediate version of the Constructive Cost Model (COCOMO) developed by Dr. Barry Boehm of TRW. SECOMO extends COCOMO by providing a mechanism to determine what portions of the life cycle activities will be performed by the Government, the type of funds that will be expended, and the portions of work which will be performed by the LCSE Center.

Originally the effort was considered to be a FORTRAN to Ada translation, but early in the conversion phase we realized that a simple translation would do little to improve the quality of the product. As a result, the project became a redevelopment effort. In addition, ten months into the project a new requirement was added; to retarget the SECOMO to the IBM PC/AT and XT compatible targets.

During the course of the development effort we had to face many of the 'hard-knocks' of any software development effort. This paper discusses our experiences and lessons learned in the area of design methodologies, Ada coding, testing, integration and retargeting.

Development Background

When the five members of our development team started this project, none of us had any experience in the development of an actual system in Ada. All of our members had some Ada programming language training. However, the only Ada programs we had developed were small individualized classroom projects. Furthermore, we did not have any training in software engineering practices, experience in the SECOMO application domain and had not worked together as a development team.

The hardware and software configuration required to develop the VAX targeted SECOMO version was as follows:

VAX Host Development Environment
- VAX 780 with VMS Operating System,
- 16 Megabytes of Memory,
- 500K bytes of disk space,
- DEC Ada Compiler versions 1.2 and 1.3 with Symbolic Debugger,
- VMS Environment Tools,
- VMS Ada Language Sensitive Editor,
- ACS Ada Program Library Management Utility.

Development Experiences

The SECOMO User's Guide and the executable FORTRAN programs were used to analyze the requirements of this system. We decided to maintain the FORTRAN version's formats for the various user's inputs, reports and datafiles. In addition, the FORTRAN source code provided us with all of the COCOMO and SECOMO algorithms. The 'Software Engineering Economics' textbook by Barry Boehm was reviewed for any additional COCOMO information.

At the start of the project we decided that SECOMO would be coded entirely in Ada. The textbooks 'Software Engineering with Ada' by Grady Booch and 'Programming in Ada' by J.G.P. Barnes were used as references and guidelines for software design and coding.

We decided to use functional decomposition to develop SECOMO. Our first decision was to decompose the model into two separate modules: the data input portion, SECOMO_1; and the report output portion, SECOMO_2.

Our team understood the requirements of SECOMO_1, however some of the requirements for SECOMO_2 were not fully analyzed. We were very anxious to code in Ada. Therefore, we decided to develop SECOMO_1 first and not worry about SECOMO_2 until SECOMO_1 was completed. We did not think this decision would cause a problem.

During each development, the two main modules were further decomposed into smaller modules. The SECOMO_1 module was decomposed based on the screen format functions that captured various types of user input information: datafiles, systems, activities and cost drivers. The SECOMO_2 module was decomposed base on the following functions: user inputs, development calculations, maintenance calculations and report outputs. In both development phases, the interface requirements between the module were defined. When this process was completed, the design and coding of each module was assigned to a member of our team.

During the design process we represented the functionally decomposed modules as Ada packages. Members of our team used an objected oriented approach for designing each of the packages. First, we described each function as a paragraph of text. Next, we identified noun phrases as abstract types and objects and the verb phrases as operations. Our main design goal was to design a packages that hid the implementation from higher levels. We did not consider the identification of reusable components during the design phase.

Our coding and unit test strategy was to first code and compile the package specifications derived from our design process. All visible package specific abstract data types, data objects and subprogram specifications within the package were compiled. Next, we coded and compiled the package body using the separate clause for the subprogram bodies. This allowed specifications of each subprogram to be directly visible, but bodies could be separately compiled and therefore textually and logically hidden. Each member of the team coded and tested within a separate Ada program sublibrary. This proved to be of great value for the management of the coding effort.

Our strategy enabled each team member to concentrate on good structure without worrying about the implementation of the subprograms. After the overall structure was established, each team member started coding the subprogram bodies. Each subprogram was then tested and integrated within the package. Each package was then integrated and tested in a 'system build' designated library that checked interfaces. Reusable components in our implementation were flushed out during the integration process. Some new packages were built that incorporated reusable types and subprograms.

Development Lessons Learned

In retrospect we can look back on this effort and see what mistakes were made, see what things we should have done differently and evaluate what we have learned. Some of the lessons we feel are important to pass on to others about to embark on this same type of effort are:

Due to our impatience to code in Ada we fell victim to a common beginner's trap of not looking at the complete design. When our team began the SECOMO_2 development, we realized several access and record types that had been designed for the SECOMO_1 program could have been better designed for the SECOMO_2 implementation. Some objects and types developed in SECOMO_1 had to be changed or tailored for the SECOMO_2 implementation. This would not have happened if we had designed the entire model before we started coding.

We also should have made a more concerted effort to identify reusable code during the design phase. Once the code was developed, we kept evaluating the code for possible reuse. We would group the potential reusable software functions into common packages that could be used across all modules. However, this approach was not efficient as code needed constant modification.

On the positive side we took full advantage of DEC's Ada library facility for shared libraries and sublibraries and the Ada separate compilation capability. Each functional package was developed in it's own sublibrary which would consist of all the logically related entities needed to implement and test each SECOMO function. This allowed each of us to completely code and test packages without affecting the code of the other individuals on the team.

Retargeting Background

Upon completion of the effort to convert the SECOMO program into Ada, we were tasked to retarget the code for the IBM PC/AT and XT. This effort had two primary objectives. First to produce IBM PC/AT and XT executable versions of the program and second to acquire knowledge about the Ada features which need to be addressed during a retargeting effort.

Two of the original team members had left the organization before this task began and one new member had joined the team. None of the team members had had any experience working with IBM-PC hardware, the MS-DOS operating system, the Alsys compiler, the Alsys AdaProbe debugger or the communication programs for transferring files to the PC's from the VAX.

The following hardware and software configuration was used during the retargeting effort:

VAX Host Development Environment
- as described above

IBM PC/AT Environment
- Intel 80286 processor,
- Intel 80287 floating point coprocessor,
- 640K bytes extended memory,
- 30Megabytes disk storage,
- Alsys compiler versions 1.3 and 3.2 with AdaProbe

IBM PC/XT Environment
- Intel 8086 processor,
- Intel 8087 floating point coprocessor

Retargeting Experiences

We began this effort with the assumption that retargeting SECOMO would be a simple task. We believed Ada was portable. However, differences in operating systems, compiler capabilities and hardware dependent features slowed our effort.

The approach we took was to maintain a single baseline of the source code. All modifications to the code would be performed on the VAX and then transferred to the IBM PC/AT for compilation. During the initial development effort, we had documented the target dependent features within the Ada source code. We made the necessary changes to the identified target dependent features, transferred the code to the PC/AT and started compiling.

Retargeting Lessons Learned

Due to memory and storage limits imposed by the target processor, it became necessary during the retargeting effort to restructure the code. Although these limitations were clearly specified in Appendix F of the Alsys User's Guide, we had not considered them because the retargeting effort had not been specified prior to the first development effort. If we had known of the retarget effort, we would had made the necessary adjustments within the original implementation.

The first problem we encountered was that the maximum size of a dynamic record object was exceeded during execution of SECOMO_2. This problem was caused by a constraint due to a hardware limitation. Upon examination of the code, it was determined that by splitting the single record up into several smaller records we could solve this problem. This change was relatively simple.

The second problem we encountered was when we attempted to retarget on the IBM PC/XT. We simply ran out of memory when SECOMO_2 was executed. The executable image on the AT was produced using the I286_Protected mode, allowing the dynamic data to exceed the 640K bytes of address space. Since this was not possible on the XT, we once again had to restructure the source code. We examined the code and found that SECOMO_2 had 'withed' many of the SECOMO_1 packages. We did this to take advantage of several subprograms within each package. We knew that only some subprograms within the SECOMO_1 packages were required.

Therefore, some subprograms were placed in a separate package that could be 'withed' by both modules. This eliminated SECOMO_2's need to 'with' all of SECOMO_1's extra baggage, resulting in an executable image nearly 50% smaller, which executed on the XT.

The resolution of both these problems highlights the importance of the use of Appendix F of the compiler reference manual. This appendix describes all implementation-dependent characteristics of the compiler and target processor. It should be referenced during the design phase of the system development in order to eliminate the problems mentioned above. However, these problems made our team realize how easy it is to restructure Ada code.

Conclusion

The final product is a system which contains 25 packages and 300 subprograms. The team developed a quality product which has been transitioned to the Center's support contractor for maintenance. They have stated that they are able to make updates to the model more easily and quickly than they had been able to while maintaining the FORTRAN implementation. In addition, it should be noted that the total lines of source code was reduced from approximately ten thousand lines of FORTRAN to eight thousand lines of Ada.

All of the team members improved their software development techniques and learned the importance of utilizing modern programming methodologies. Furthermore, we gained knowledge and skills in Ada programming and more clearly understand the advantages Ada provides the software developer and maintainer. Finally, we believe we are in a better position to assist the Center's Project Leaders and CECOM's Program Managers because of the work performed during this project.

Acknowledgements

Special thanks are due to Mr. Mark Miller now with Simmons Precision, Verginnes, VT and Mr. Larry Persie now with Singer, Kearfott Division, Little Falls, NJ. Both of these individuals contributed substantially during the initial development effort.

References

Barnes, J.G.P., 'Programming in Ada', 1984, (Addision-Wesley Publishing Co.)

Boehm, Barry W., 'Software Engineering Economics', 1981, (Prentice-Hall, Inc., Englewood Cliffs, N.J.)

Booch, Grady, 'Software Engineering with Ada', 1983, (The Benjamin/Cummings Publishing Co., Inc., Reading, MA)

Ada Programming Language, ANSI/MIL-STD-1815A, U.S. Government, 1983

Developing Ada Programs on VAX/VMS, Digital Equipment Corp., Maynard, MA, 1985

SECOMO Instruction Manual, U.S. Material Command, Management Engineering Activity, Huntsville, AL, June 1985

SECOMO User's Guide, IIT Research Institute, Sept. 1985

Authors

John T. LeBaron is a Computer Scientist with the Center for Software Engineering, Advanced Software Technology, Fort Monmouth, N.J. He received his MS in Computer Science from Kean College. He is currently working in the Software Engineering Technology Activity, Life Cycle Process Funtional Area investigating software development environments.

Raymond J. Menell is a Computer Scientist with the Center for Software Engineering, Advanced Software Technology, Fort Monmouth, N.J. He received his BS and MSCS from Monmouth College, N.J. and is currently pursuing a MS in Software Engineering at Monmouth College. He is currently working in the Software Engineering Technology Activity, Life Cycle Process Functional Area investigating life cycle processes and software case studies.

Judith D. Richardson is a Computer Scientist with the Center for Software Engineering, Advanced Software Technology, Fort Monmouth, N.J. She received her B.S. in Computer Science from the University of Maryland-College Park. She is currently working in the System Software Technology Activity, Ada Technology Functional Area, investigating Ada interface to SQLs and GKSs, and Ada semantic definition.

Johnny Ng is an Electronic Engineer with PM-Joint Star, Ft. Monmouth, N.J. He received his B.S. in Electronic Engineering from City College of New York. He is currently working in the Technical Management Division.

Requirements Engineering and Ada

George E. Sumrall

U. S. Army Communications Electronics Command
Center for Software Engineering
Fort Monmouth, New Jersey

Abstract

The quality and cost of developing Ada-based software is significantly affected by activities and decisions which precede the software life cycle, particularly in the area of requirements definition. This paper examines some of the difficulties associated with traditional viewpoints and ways of defining and handling requirements. Some new perspectives and fundamental concepts are introduced including the notion of user viewpoint and associated interfaces, requirements evolution and requirements engineering. A new perspective concerning the role of requirements engineering, system engineering and software engineering across the system life cycle is proposed. Finally, an approach to the handling of requirements in Ada-developed software is proposed.

Introduction

Use of the Ada programming language and software engineering methods which Ada supports hold great promise for being able to reduce the cost and time required to develop and maintain software for computer based defense systems. Unfortunately, Ada and software engineering methods alone are not sufficient to ensure that promise. One source of difficulty is in the way that requirements are handled for Ada-developed software .

Software requirements (which presumably are derived from some sort of system engineering process) are typically viewed to be **"The Requirement"** (with the expectation that "The Requirement" is complete, specifying exactly what the software is to do, and that it will not change). Accordingly, "The Requirement" is often written into the software (or system) development contract. It is not uncommon for the Ada programmer to discover that "The Requirement" is not complete. However, he or she often is constrained by a deadline and "picks" a solution without taking the time to research what the customer wanted. It is also not uncommon, when the system is delivered many months (or years) later, for the customer (user) to discover that the system does not perform as he "wanted". It is also not uncommon for the customer (often the Government) to want to change "The Requirement" during development. Because it is written into the contract, such changes are very expensive and the customer is forced to accept an unsatisfactory product or fund excessive cost overruns. Clearly, these problems are not Ada-based, yet they can have a significant effect on the cost and quality of Ada-based software.

This paper takes a look at the traditional way of viewing and handling software and system requirements and provides some new approaches and insights concerning requirements as they relate to the creation and evolution of Ada-developed software.

It is noted that although the motivation for this work stems from concerns in the development and evolution of Army tactical embedded systems, the concepts and techniques discussed should be applicable across a wide range of application domains.

The Traditional Approach

The following diagram depicts a traditional life cycle approach for the development of a software-critical system.

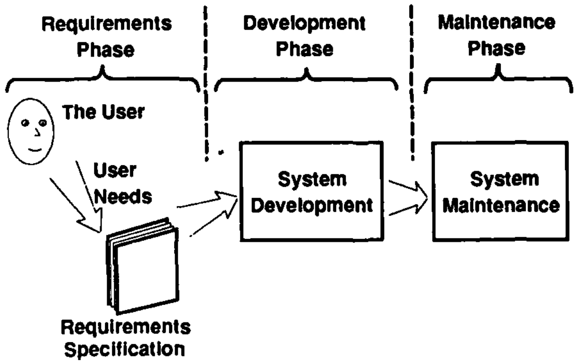


Figure 1 - Traditional System Life Cycle View

From the viewpoint of the software designer or programmer, *the user* represents some person or organization who will be the eventual user and/or owner of the system being developed. The user's needs for the system are somehow known and captured in the *requirements specification* document. The software development process is part of the *system development* activity and typically follows a "waterfall" life cycle model similar to that shown in figure 2 [1] [2].

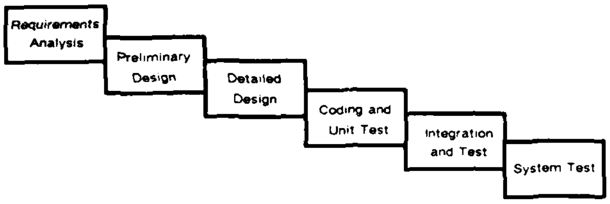


Figure 2 - Software Development Life Cycle

This life cycle model presumes that the software requirements are known at the start of the process. The software requirements are somehow derived from the *requirements specification* as part of the system engineering process.

The end of the *development phase* (see Fig. 1) occurs when system development is completed and the system is delivered to the user.

The *maintenance phase* is viewed as the "period in time" when software problems are corrected and perhaps the system is made "nicer to use". This phase is often viewed by the software developer or programmer as somebody else's problem. Supported by this view, difficult software problems which are encountered during development are sometimes "swept under the carpet" and left for somebody else to solve during the maintenance phase.

With this (somewhat simplistic) traditional view of the system life cycle in mind, lets proceed to look deeper into some of the concepts, expose some myths and explore some new thoughts concerning the concept of *requirements* throughout the system life cycle.

Some Myths and Some New Ideas

In the discussions to follow, an attempt has been made to group related requirements issues. New viewpoints and concepts are highlighted in ***bold italics***.

1. The User

In reality, "the user" is not just one user, but several different users[1]. In the case of an Army tactical command and control system for example, several users come to mind: an operator (enlisted person), the commander and a maintenance person to name three.

Each of these users has a different "user viewpoint". For example, the commanders view of and interaction with the system is certainly different than that of the operator and similarly, the maintenance person.

Associated with each user viewpoint is a unique, abstract "user interface" which characterizes and identifies how that user interacts with and views the system. This interface applies not only to the actual "man-machine" interface in the target system, but applies to any prototyping and documentation associated with the system (e.g., a user manual).

The notion of "user" can be extended to non-human entities such as devices connected to communications ports or data links. Equipment connected to these ports are "users" of system resources, must interact with the system in some manner and in fact have their own unique interface. (We already recognize this fact for some types of data communications interfaces.)

The "user requirements" for a system comprise the union of the requirements of the different user viewpoints.

2. Requirements Drivers

System requirements stem from several sources:

a. Need tor new or improved capabilities
b. Emergence of new technology and products
c. Familiarity with the current system
d. Perceived new "threat" to the user
e. New level of user sophistication
f. Solutions to previous requirements

There are three primary factors which drive requirements for a system. Their relation is shown in Figure 3:

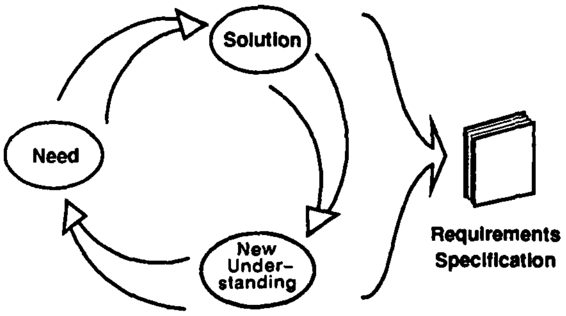


Figure 3 - Need-Solution-Understanding Relation

The awareness of a need results in a solution being sought. Realization of the solution can result in new understanding and/or thought patterns which can trigger the awareness of a new need. The system requirements can be driven by any or all of these factors.

[1] This point may not apply in the case of single user personal computer software.

By extending this concept further, one can reason that in some cases, ***the introduction of a new system (i.e., capability) into an organization can change the way the people in the organization operate and do business***. For example, the introduction of a new command and control capability into the Army battlefield or into a corporate headquarters can change the way the commander and the corporate executive operate and "do business". ***Even though this process may obsolete the new system, it is a necessary cycle to go through.*** Furthermore, it is probably impossible to fully predict the impact of the introduction of such a new capability.

By taking a different look at the need-solution-understanding diagram, one can construct a *requirements evolution tree*:

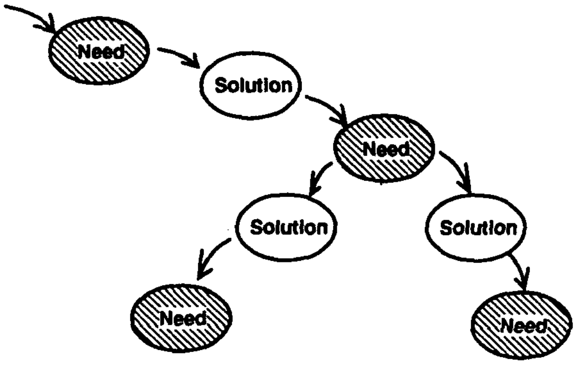


Figure 4 - Requirements Evolution Tree

The idea here is that a solution to one need (i.e., requirement) may generate a new need. The solution(s) to that need may in turn generate a new need. Conversely, technology advances may currently be available which could satisfy a prior need and thus obviate the current need. ***Thus in dealing with requirements, one needs to be aware of the evolutionary nature of requirements.*** In particular, requirements analysis activities should include "looking back up the tree" to examine earlier needs and solutions to see if current understanding and technology applied to an earlier need might be a better solution than the one currently being sought.

3. The Nature of Requirements

Software developers and programmers often view system and *software requirements specifications as an* authoritative, unquestionable statement of what the system is to do and how it is to behave and perform. The requirements document is expected to be an accurate, complete and consistent statement of the user's needs and is expected not to change.

Experience on the otherhand, does not support that viewpoint. Requirements have changed, even during development. Inconsistencies have been found. Designers routinely must make design decisions which affect system behavior in areas not covered by the requirements specification.

The following conclusions have been reached:

A complete, consistent and accurate statement of requirements for a system may be impossible. There are several reasons for this: a) inability of users to foresee all level of detail, b) complexity of the system and c) inconsistency between various user viewpoints.

For some systems, a complete, consistent and accurate statement of requirements is impossible. This is especially true for systems where the introduction of a new system into an organization will change the way the people in the organization operate and do business. An example is an Army command and control system.

Requirements will change and may change often [3]. This conclusion follows from the two previous conclusions. It is supported by experience.

4. Requirements vs. Needs vs. Desires

We have all seen cases where what appeared to be an absolute and essential requirement was relaxed or waived when the user (buyer) was presented with the cost of such a feature. Was the original requirement a real requirement or a desire? The answer can be found by looking at the notion of *essentiality of requirements.* For each requirement in a requirements specification, assign a Requirements Essentiality Factor (REF) such that

for REF = 1, the requirement is absolutely essential
and for REF = 0, the "requirement" is not desired.

An *absolutely essential requirement* is defined to be one that the user must have, else the system is not wanted.

For all requirements, $0 < REF <= 1$
and usually $REF < 1$.

Furthermore, the REF is a dynamic factor which is dependent on other requirements, their associated REFs, the user's needs and the users financial resources.

In other words: ***Few requirements are absolutely essential. The essentiality of the rest vary and are negotiable.***

Requirements Development Process Considerations

The concept of defining requirements is not new in engineering and architectonics. The process is mature and works rather well in many areas. The following example gives some insight into how requirements are developed for a new building and suggests an analogous approach as to how requirements for software-critical systems could be developed.

Several years ago, I chaired a requirements committee for a church who wanted to build a new church building. (The old building had burned down.) The committee consisted of persons representing each organization in the church and the staff (i.e., different user viewpoints). After hiring an architect, we met to discuss our requirements for the new building. In particular, after reviewing our mission and purpose, we identified the functional requirements which would be needed in the new building to enable us to accomplish our mission. The requirements statement which was developed was agreed tc by all user viewpoints. A concept of the form of the building was discussed and an initial plan (conceptual model) was developed by the architect. Based on feedback from the committee (i.e., users), the plan was revised more than once. At one point in the process, the architect's cost estimate was almost twice the planned budget. Requirements were of course scaled down until an affordable plan was agreed to and accepted. Several different models (reflecting different user viewpoints) were useful in achieving consensus. The architect then prepared a detailed design and architectural drawings. It should be noted that up to this point in time, the architect's role was twofold: he served as a consultant to the user(s) and acted as a surrogate for the (yet unselected) building contractor and subcontractors. The architect then served as a consultant to help us select a builder. As construction proceeded, the architect oversaw this effort, acting as a surrogate for the church (users).

With the building architectural model in mind and using some of the new concepts presented above, consider the following model for the requirements development process.

A Proposed Requirements Development Process (for Software-Critical Systems)

The key player is a *System Architect* who has a role similar to that of the building architect described above. During requirements development, he serves as a surrogate for the system and software developers. During system development, he serves as a surrogate for the user viewpoints.

The *System Architect* and representatives of the various user viewpoints, who have authority to make decisions concerning system requirements[4][5], meet along with representatives from the development organization[2] to discuss system capabilities and requirements. Prototyping and similar techniques are used to help achieve a consensus on system capabilities, functionality, performance and interfaces. The requirements for the system are documented in the *System Requirements Statement* (see below) which must have working group consensus and approval of the parent organization(s) of the requirements working group. Next a *System Specification* is prepared by the System Architect which also requires consensus and approval. The System Specification is then given to the development organization[2], who selects a development contractor and with the help of the system architect, oversees development of the system.

The System Requirements Statement

The System Requirements Statement (SRS)[3] is prepared by the requirements working group with the assistance of the System Architect. The SRS identifies the purpose of the system and provides an operational scenario of how the system is to be used. It contains a description of the functional behavior of the system, interfaces (human and other), performance characteristics, constraints and special considerations such as security. Statements concerning the "ilities" (e.g., maintainability, portability, etc.) might also appear. The level of detail is determined by the working group. There is obviously a tradeoff here between amount of detail, volume of text and understandability. A short SRS is probably preferable. All absolutely essential requirements should be so identified. The other requirements should have a relatively high Requirements Essentiality Factor (REF). The SRS then becomes an agreed to statement of what the users want the system to be.

The System Specification

The System Specification (SS) is prepared by the System Architect. Several iterations of the SS are envisioned. Each version is fed back to the user working group (ideally recognizing and using the various user viewpoint interfaces). The system development organization participates in these reviews. The System Specification development may cause the system requirements to change. If so, the SRS should be updated accordingly.

At some point in time, the SRS and SS become agreed to by all user viewpoints, the System Architect and the developing organization. Upon organizational approval, these documents should be put under configuration management since they may change throughout the remainder of the life of the system.

[2] The *development organization* is the organization who will have system development responsibility.

[3] Not to be confused with the IEEE's Software Requirements Specification, also abbreviated SRS [6]

Some Observations

1. The term *Requirements Development* was used above to describe the process of producing the System Requirements Statement(SRS). Under the model proposed, the requirements statement is indeed developed through a disciplined process and put under configuration management. The process has a life cycle. Further, the SRS is an architected product[7]. Thus the concept of *Requirements Engineering* is a valid notion[7] and is defined as follows:

Requirements Engineering is a systematic approach to the development, transition, evolution and dissolution of requirements.

2. During the past year or so, I have sought answers to the questions "*What is a requirement?*" and "*How would I know one if I encountered it?*". The answer can now be given:

A requirement is some capability or feature of the system that is stated in the System Requirements Statement.

Requirements Engineering in the System Life Cycle

The traditional system development model shown in figure 1 suggests that requirements development stops at the end of the requirements phase. Based on the ideas presented here, it appears that requirements engineering activities continue throughout the whole system life cycle. Requirements engineering is probably most intensive at the start of the life cycle. The nature of the requirements engineering activities may change during the life of the system, but never cease.

Similarly, system engineering activities (including software engineering) are needed across the whole life cycle. Figure 5 shows the relative amount of these activities across the traditional life cycle phases.

This model shows *requirements engineering*, system engineering and software engineering activities co-existing in all phases of the system life cycle. Ways must be found to separate the concerns of each discipline while at the same time taking advantage of their co-existance. Boehm's spiral model[10] is one approach that accomodates activities in these disciplines.

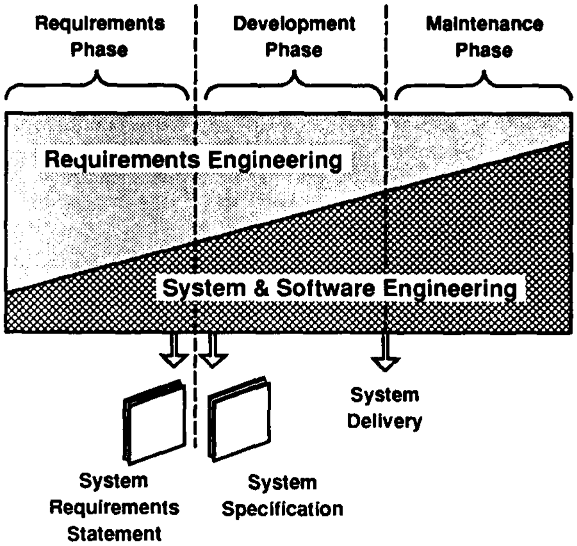


Figure 5 - New View of the System Lifecycle

Role of the Ada Language

There are several ways that the Ada language can be used in developing and handling system and software requirements.

1. Probably the single most important capability that Ada offers for handling requirements is the ability to modularize and encapsulate software requirements. Using principles of information hiding and modularization developed by Parnas [8] [9] and others, software requirements can be structured, modularized and encapsulated using Ada package(s). This should protect the software system from any adverse affects due to changes in requirements . It is envisioned that these Ada packages would not have package bodies.

2. A second capability which Ada offers is the capability to write reusable Ada packages defining common interfaces. Such interfaces include communications protocols, database (SQL) interfaces and graphics standards such as GKS. Further application unique interfaces should be encapsulated in Ada packages. These packages could have package bodies.

3. It is possible that Ada could play a role in helping to develop requirements prior to system specification through prototyping and modeling of various user interfaces and by providing the user(s) a "feel" for the system functionality. It will probably be some time for this role to become a reality, since the development of standard interfaces and common interfaces as well as an Ada-based prototyping capability will need to be achieved.

4. The point was made earlier that it is probably impossible for the system requirements statement and hence the software requirements document to completely state all of the requirements. Hence, it is expected that designers and Ada programmers will routinely have to make design decisions that affect the behavior or performance of the system with no guidance from any requirements document. Procedures need to be established as part of the design and coding process to document these decisions and to surface them in the software management process, possibly at design and coding reviews.

Conclusion

This paper has attempted to point out some of the problems related to the impact of the requirements development process on the development of Ada-based software. Obviously a lot of work remains to be done to turn these ideas into workable procedures. Software engineers and programmers can not trust that someone else will solve the "requirements problem". It will take the concerted effort of software engineers, system engineers and requirements developers to achieve a workable solution to this problem.

Acknowledgment

I would like to thank Dr. Serafino Amoroso for recognizing the concept of user viewpoint and for bringing it to my attention. I am especially grateful to Tom Wheeler for his encouragement and engaging discussions on the subject of this paper.

George E. Sumrall has been an electronics engineer with the US Army Communications Electronics Command (CECOM) at Fort Monmouth, New Jersey for the past 30 years. He has played an active role in Army and DoD software technology development for the past 10 years. He has chaired the DoD Methodology Coordination Team under the auspices of the Ada Joint Program Office and later, the STARS Joint Program Office. He is currently Chief of the Life Cycle Process Activity of the Office for Advanced Software Technology within the Center for Software Engineering at CECOM. He has a MS in Electrical Engineering,

References

[1] DOD-STD-2167, "Defense System Software Development", 4 June 1985

[2] W. Royce, "Managing the Development of Large Software Systems", Proceedings, IEEE WESCON, August, 1970

[3] E. Schlosser, "Implications of Ada for Requirements Analysis and Design", Presentation at ACM SIGAda, January 19, 1987, Hollywood, FL

[4] G. Rush, "A Fast way to define system requirements", COMPUTERWORLD, October 7, 1985

[5] D. Leavitt, "Team Techniques in System Development", Datamation, November 15, 1987

[6] IEEE Std 830, IEEE Guide to Software Requirements Specification, 1984

[7] R. DeMillo and G. Sumrall, "Workshop on Future Development Environments", ISTAR III Conference on Information Mission Area Productivity, Department of Army, DISC^4.

[8] D. L. Parnas, "Information Distribution Aspects of Design Methodology", Proceeding of IFIP Congress, 1971

[9] D. L. Parnas, "On Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, December, 1972

[10] B. W. Boehm, "A Spiral Model of Software Development and Enhancement", SIGSOFT Engineering Notes, Vol. 11, No. 4, Aug 86

TOWARD AN OPERATIONS-ORIENTED METHODOLOGY FOR Ada® REAL-TIME SYSTEMS

Richard F. Vidale[*] and Bard S. Crawford[†]

[*]Boston University
Boston, Massachusetts

[†]The Analytic Sciences Corporation
Reading, Massachusetts

ABSTRACT

Essential requirements of a real-time Ada development methodology and an operations-oriented approach to satisfying these requirements are outlined. This approach is designed to capture performance requirements early in the life cycle and trace them through to validation. A graphical technique featuring SRCs (Stimulus Response Charts) and SODs (System Operation Diagrams) is featured. An example problem illustrating the use of the approach, including a key fragment of resulting Ada code, is included. Characteristics of an appropriate graphics workstation support environment are briefly discussed.

1. INTRODUCTION

This paper summarizes the results of an inquiry[‡] into the design of Ada real-time systems in which performance issues are dominant. The inquiry was motivated by our observation that available Ada-specific design methodologies, namely Object-Oriented Design [ABBO83], [BOOC83], System Design with Ada [BUHR84], and Process Abstraction Method for Embedded Large Applications [CHER85], lack the means to explicitly capture performance requirements and implement them in a controlled manner. Methodologies with this capability are clearly needed if Ada is to be successful in real-time applications.

As our study progressed, we became convinced that an operations-oriented approach was the most direct way to deal with performance requirements. We were strongly influenced by Alford [ALFO85] and McCabe [MCCA85], who maintain that performance issues must be addressed early and continuously throughout the life cycle. By the end of the inquiry we had outlined a methodology, with supporting graphic conventions, which is fundamentally different than existing Ada design methodologies. We believe that further development of our preliminary results could help fill a current gap in Ada design technology.

2. REQUIREMENTS OF A METHODOLOGY FOR REAL-TIME Ada DEVELOPMENT

Before presenting a step-by-step description and example of our approach, we shall state some essential requirements of a real-time Ada development methodology and indicate which aspects of our methodology address these requirements.

The first requirement is to provide a suitable format for capturing the stimulus/response requirements of a real-time system. We use Stimulus/Response Charts (SRCs) to document the required response to each stimulus, traced back to system requirements. Each SRC captures a requirement at the level of a paragraph or sentence in the system requirements specification.

The second requirement is a structure linking the SRCs together to represent a flow of operations in which the response of one stimulus may become the stimulus of another response. We use System Operations Diagrams (SODs) to provide macroscopic representations of system operation. SODs serve as roadmaps showing the relationships of the SRCs. The process of documenting a system's performance requirements can start with SRCs, or with SODs, or with both.

The third requirement is the ability to establish point-to-point timing requirements along paths of the SODs, upon which resource allocation decisions can be based. Some of these timing requirements may be stated directly in the system requirements specifications; others may have to be derived from system-level requirements, such as throughput. In the latter case, it will be necessary to model the system operation to derive the point-to-point timing requirements. The SODs can provide a database from which a model can be developed.

The fourth requirement is a means to make a preliminary allocation of processing resources to meet the point-to-point timing requirements. In simple cases the allocation can be done by inspection of the SODs. More complicated situations require automated analysis of the system performance models as described in [MCCA85], for example.

The fifth requirement is a means to facilitate Ada task body design, once the allocation of processes to tasks has been made. We use the Mealy form [WARD85] of State Transition Diagrams (STDs) as an intermediate representation of the states of a task, each state representing the execution of a process allocated to the task. The state transition design can then be realized in the task bodies using a select statement.

The sixth requirement is that the methodology should facilitate the design of test plans. In our methodology, the flow of system operations, and their attendant timing requirements, are mapped directly onto the SODs. The development of performance test plans is then largely a matter of

®Ada is a registered trademark of the U.S. Government (AJPO)

[‡] Independent Research and Development Program, I-8807, The Analytic Sciences Corporation, June - September, 1987.

exercising specific paths on the SODs to verify the estimated delays.

3. AN OPERATIONS-ORIENTED DEVELOPMENT METHODOLOGY

The Operations-Oriented Development (OPSOD) methodology focuses on performance requirements early in the development process and traces performance in a controlled way through specification, implementation, and validation. OPSOD provides models of the system to allow the designer to make informed decisions about how to specify processors and algorithms to meet performance requirements and how to validate performance. The eleven steps in OPSOD are listed below.

Step 1 Develop a written Requirements Specification document

Step 2 Develop a set of Stimulus/Response charts (SRCs) based on the Requirements Specification

Step 3 Link the Stimulus/Response charts to form a set of System Operations Diagrams (SODs)

Step 4 Establish point-to-point (PTP) timing requirements on the SODs

Step 5 Make estimates of the processing resources (number of machine instructions) required by the operations

Step 6 Make preliminary allocations of system operations to tasks and processors, exploiting opportunities for parallelism revealed by the SODs

Step 7 Develop State Transition Diagrams (STDs) for tasks and processors to show how concurrent threads of processing are handled

Step 8 Analyze performance, based on preliminary processor allocations and STDs

Step 9 Develop Ada code to implement the STDs

Step 10 Test the Ada code, using the test plans from Step 4

Step 11 Revise the hardware/software design and/or requirements so that performance requirements are satisfied.

These eleven steps do not fit neatly into the conventional life-cycle phases of software development, i.e., those defined in [DEFE85]. The reason is that the conventional (waterfall) life-cycle model is top-down, separating specification, design and implementation concerns. OPSOD is largely bottom-up because performance requirements drive design and implementation decisions that are typically considered low-level, and these are addressed before high-level architectural decisions. Figure 1 shows a mapping between the eleven OPSOD steps listed above and the six phases of [DEFE85]. The first five steps in OPSOD correspond to conventional requirements analysis, but are much more performance oriented than usual. Steps 6 through 8 correspond to detailed design, but precede any attempt to create a high-level software architecture, i.e., the packaging of unit operations into CSCIs (Computer Software Configuration Items) and CSCs (Computer Software Components). Step 9 corresponds to the conventional top-down design steps, and Step 10 corresponds to conventional bottom-up testing. Step 11 is included in recognition that iterative development may still be necessary.

4. A SIMPLE EXAMPLE OF OPERATIONS-ORIENTED DEVELOPMENT

The following example was invented to illustrate the OPSOD methodology. It provides samples of the first nine steps listed above. Though small-scale, the example illustrates the following design issues:

- Handling concurrent, asynchronous inputs
- Parallel system responses
- Parallel process synchronization and communication
- Allocation of processing to processors
- Data abstraction
- Multi-thread processing by a process
- Ada code implementation.

Informal Problem Description

The system described below is a warning system for a U.S. Navy ship. The requirements for this system are purely fictitious. Any resemblance to real requirements is purely coincidental.

The Warning System receives signals from a radar when an echo is detected. The system must first digitally process the signal, then determine if it is a valid signal from an aircraft. If so, the signal is analyzed to determine the distance to the aircraft and whether the aircraft's signature is that of a friend or foe. If the distance is less than some specified SAFE_DISTANCE, a CLOSE_ALARM is sounded. If a foe signature is identified, then a FOE_ALARM is sounded. If both conditions obtain, then the PHALANX is activated, if it is not already so. If a friend is identified and is farther away than SAFE_DISTANCE, then the PHALANX is deactivated if it is not already so.

The requirements for sounding the alarms are so stated because it takes much longer to determine friend/foe than distance, and a too close warning is desired before a friend/foe identification.

In addition to processing radar signals and making appropriate responses, the system must allow an operator to enter a change in the value of SAFE_DISTANCE at any time except while the check for distance is in progress.

Methodology Walkthrough

Step 1:

Figure 2 shows the part of the Requirements Specification which specifies PHALANX activation/deactivation.

Step 2:

Figure 3 shows two of the Stimulus/Response Charts (SRCs) which capture part of the requirements defined by Figure 2, i.e., the cases when a foe is identified. These charts all pertain to a parent operation, PHALANX Activation, in the higher-level Radar Signal Processing operation, whose label is 12. All the labels in Figure 3 are therefore numbered 12.x to show the parent-child relationship. Each SRC in Figure 3 is classified as either an

Operation, Test, or Outcome (of a test) and is given a name. Operations and tests require some processing time, outcomes take zero time. Estimates of processing time delays can be entered in the Timing Estimate box.

Step 3:

Figure 4 shows the System Operation Diagram (SOD) for the high-level radar signal processing operations, while Figure 5 shows the SOD for the next-level PHALANX activation operation. Each node of an SOD has an SRC of the same label number and name. Figure 6 is the SOD for another function of the system, in which a human operator can enter a change in SAFE_DISTANCE.

Step 4:

The point-to-point (PTP) timing requirements are provided by the Requirements Specification (not shown herein) which specifies the following maximum delays for various threads of operation in Figure 4.

Thread 1 The maximum delay from the start (Node 1) to the detection of an invalid signal (Node 4) is 2000.

Thread 2 The maximum delay from the start to sounding the Close Alarm is 3000.

Thread 3 The maximum delay from the start to sounding the Foe Alarm is 6800.

Thread 4 The maximum delay from the start to activating the Phalanx is 7000.

Step 5:

The processing requirements for the SOD nodes are estimated as follows: For Figure 4:

Node	Machine Instructions
1	1000
2	500
5	1000
6	5000
11	10
12	variable
13	10

For Figure 5:

Node	Machine Instructions
12.1	30
12.6	20
12.12	10
12.13	10

For Figure 6:

2	10

Step 6:

For simplicity, assume that a machine instruction takes one unit of time to execute. Analysis of Figures 4 and 5 shows that the timing requirement for thread 4 cannot be met with a single processor, since it takes 60 units of time to traverse Phalanx Activation (Figure 11) and thus 7580 to traverse Radar Signal Processing (Figure 10), even without changing safe distance.

Preliminary allocation of system operations is made by assigning operations to three processors (Figures 7 and 8). Note that operations of two different SODs are assigned to the DISTANCE_ANALYZER processor.

Step 7:

A State Transition Diagram (Figure 9) shows how the two threads of processing are handled by DISTANCE_ANALYZER.

Step 8:

We now see that the maximum delay through Node 5 will be 1010, when the analysis of distance must wait a maximum of 10 units during which SAFE_DISTANCE is being updated. Then the estimated thread processing times become:

Thread	Est. Processing Time	Maximum Delay
1	1500	2000
2	2520	3000
3	6510	6800
4	6560	7000

These processing times assume the three processes are each being executed on their own dedicated processors, and the communication/synchronization overhead is negligible.

Step 9:

An Ada program fragment (Figure 12) shows how the state-transition mechanism of Figure 9 is implemented with a select statement in the body of DISTANCE_ANALYZER. It is assumed that the target machine has multiple processors and that it is possible to specify that each of the three processes will run on its own dedicated processor.

5. IMPLEMENTATION OF OPSOD

Any development methodology, to find acceptance, must be cost-effective. It must be easy to learn and easy to use, and must shorten development time. We believe our Operations-Oriented Development methodology will prove cost effective if implemented in a workstation environment. The key to such an implementation is the creation of a central data base as information is entered through OPSOD's charts and diagrams.

The technology for creating databases from graphical and tabular input is well known. It should not be difficult to relate the data on the Stimulus/Response Charts to data on the System Operations Diagrams. The implementation software should adhere to the principle of enter-the-data-once. For example, if a response on a Stimulus/Response Chart is a stimulus on another SRC, the arc joining the corresponding nodes of the System Operations Diagram should be automatically generated. If an arc between two nodes on the SRC is entered, an as-yet-unspecified postcondition item is created on the SRC of the originating node and a corresponding precondition item is created on the SRC of the terminating node. In addition to reducing data-entry effort, the enter-the-data-once principle reduces the possibility of entering inconsistent connections on the SRCs and SODs, and reduces the effort required to capture a change in the system requirements. Furthermore, creation of the SODs can start with either the SCRs or the SODs, or with both in parallel, depending on which view of the system requirements is a more natural starting point.

Consistency and completeness checking must be provided. Preconditions without corresponding postconditions should be flagged and reported when a consistency check is requested. Consistency checks of stimulus/response charts have been shown to be effective in detecting incomplete or inconsistent requirement specifications [DEUT87].

From the completed database, point-to-point timing information can be automatically calculated between any two specified points on an SOD. When branching occurs within any nodes along the path between the two points, worst-case or expected delays can be computed (if branching probabilities are provided). These timing calculations will be needed frequently during the allocation of tasks to processors and the development of test plans.

If the state-transition diagrams are included in the data base, it would be possible to automatically generate control skeletons for the task bodies which implement the state-transition diagrams. Simple state transitions, as in the above example's DISTANCE ANALYZER task, are implemented by a select statement, in which each branch execution causes the task to enter a new state. In the more general case, the state transitions are controlled by guards which are functions of the states, or by a case statement using the state as the selector. As the current state is left, the state variable is updated to the appropriate next state. Thus a standard programming paradigm exists for task bodies which participate in multiple-thread control, and therefore multiple-mode (state-transition) behavior.

6. CONCLUSIONS

The Ada technology needed to fully utilize an operations-oriented design methodology is not yet available. At present, real-time Ada software designers cannot guarantee that timing constraints will be met because of the non-determinacy of the Ada language, e.g., the implementation of delay and select statements. Furthermore, the lack of distributed processing prevents designers from meeting timing requirements by dedicating processors to certain tasks. But an OPSOD approach could prove helpful now in two types of applications:

(i) In "soft" real-time systems, where timing deadlines can fail to be met without the system failing: An example is a packet-switching network, where a packet lost because of excess time delay can be retransmitted. Here, performance degrades, but the system does not fail. In these types of applications, models of system behavior, derived from the OPSOD data base, could give early indications of whether system performance will be adequate. When distributed Ada becomes available, OPSOD can be used to rationalize the allocation of processors to Ada tasks.

(ii) In "hard" real-time systems, where a missed deadline can cause system failure, run-time services outside the Ada language are used to achieve tighter control of (non-Ada) tasking: By going outside the Ada tasking mechanism, it is possible to guarantee that absolute timing requirements are met, provided sufficient power is available from the processor. Point-to-point timing information from the System Operations Diagram can give early indications of processing requirements.

Our inquiry has produced a preliminary concept of an operations-oriented methodology that is missing from current Ada design practice. We share the view of Alford [ALFO85] that the design of real-time systems should be requirement driven, and believe that our preliminary OPSOD is a correct first step toward a suitable operations-oriented development methodology for Ada. As Ada implementation technology improves, the utility of an OPSOD will also improve. As it matures, we expect operations-oriented approaches will become essential adjuncts to current Ada design methodologies.

REFERENCES

[ABBO 83] Abbott, R.J., "Program Design by Informal English Description," Communications of the ACM, Vol. 26, No. 11, November 1983.

[ALFO 85] Alford, M., Ch. 2 in Distributed Systems Methods and Tools for Specification, M. Paul and H.J. Siegert, ed., Lectures in Computer Science, #190, New York: Springer-Verlag, 1985.

[BOOC 87] Booch, G., Software Engineering with Ada, 2nd Ed., Menlo Park, CA: The Benjamin/Cummings Publishing Company, 1987.

[BUHR 84] Buhr, R.J.A., System Design with Ada, Englewood Cliffs, N.J.: Prentice-Hall, 1984.

[CHER 85] Cherry, G.W., and B.S. Crawford, "The PAMELA Methodology," Thought Tools, Inc., Reston, VA, November 1985.

[DEFE 85] "Defense System Software Development," Dod-STD-2167, June 1985.

[DEUT 87] Deutsch, M.S. and R.W. Jensen, "Real-Time Software Systems; Analysis, Validation and Management," EFDPMA Seminar, Montreal, Quebec, June 25-26, 1987.

[MCCA 85] McCabe, T.J. et al, "Structured Real-Time Analysis and Design," IEEE Ninth International Computer Software and Applications Conference, Chicago, Illinois, October 9-11, 1985.

[VIDA 86] Vidale, R.F., "Extending Object-Oriented Ada Design Methodology," Boston University, June 30, 1986.

[WARD 85] Ward, P.T., and S.J. Mellor, Structured Development for Real-Time Systems, Englewood Cliffs, N.J. : Prentice-Hall, 1985.

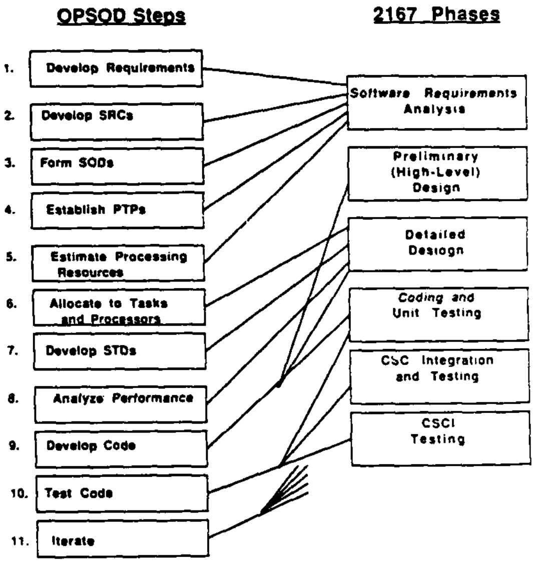


Figure 1 Mapping OPSOD Steps to DoD-STD-2167 Phases

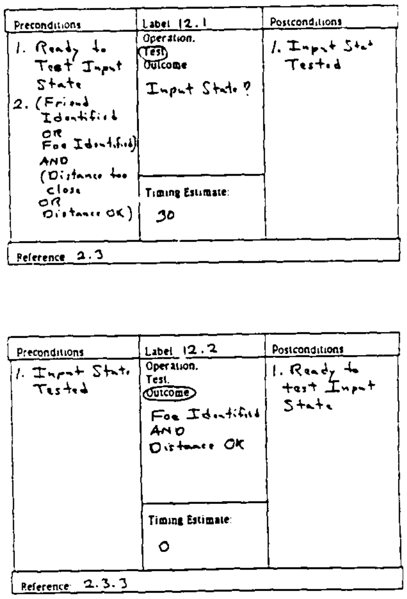
2.3 Phalanx Activation

2.3.1 The Phalanx is activated if it is not currently active and a foe is identified and the distance is less than the current value of safe distance.

2.3.2 The Phalanx is deactivated if it is currently active and a friend is identified and the distance is not less than the current value of safe distance.

2.3.3 No action is taken for all other conditions.

Figure 2 Sample Requirements Specification



Preconditions	Label 12.1	Postconditions
1. Ready to Test Input State 2. (Friend Identified OR Foe Identified) AND (Distance too close OR Distance OK)	Operation. [✓ Test] Outcome Input State ? Timing Estimate: 30	1. Input Stat Tested
Reference 2.3		

Preconditions	Label 12.2	Postconditions
1. Input State Tested	Operation. Test. [✓ Outcome] Foe Identified AND Distance OK Timing Estimate: 0	1. Ready to test Input State
Reference 2.3.3		

Figure 3 Sample Stimulus/Response Charts

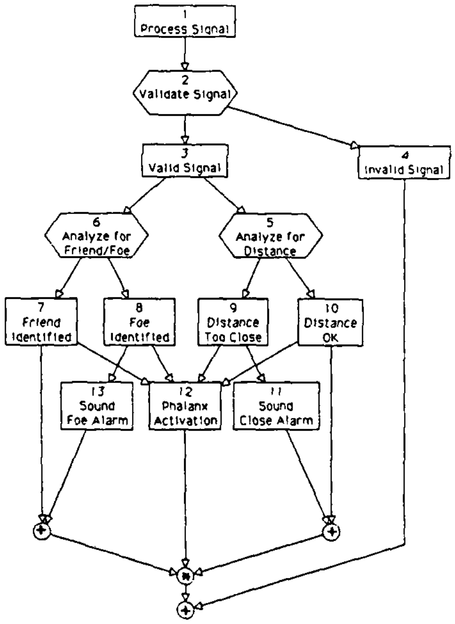


Figure 4 System Operations Diagram (Radar Signal Processing)

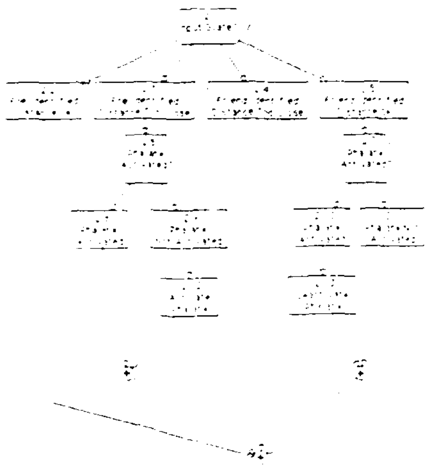


Figure 5 System Operations Diagram (12 - Phalanx Activation)

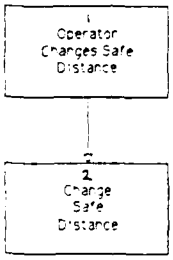


Figure 6 System Operations Diagram (*Changing Safe Distance*)

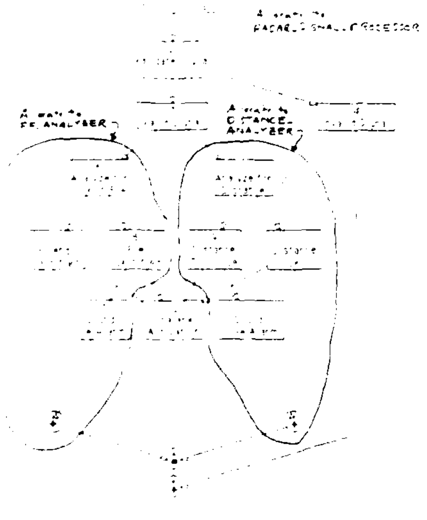


Figure 7 System Operations Diagram (Radar Signal Processing)

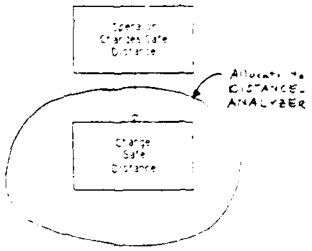


Figure 8 System Operations Diagram (*Changing Safe Distance*)

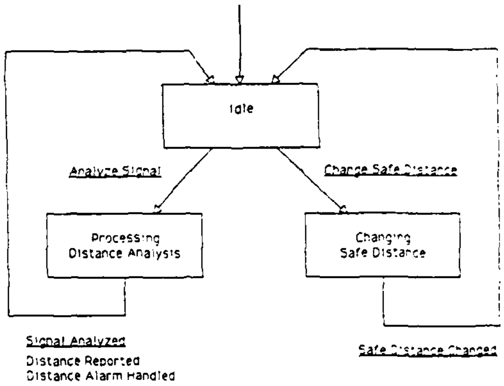


Figure 9 State Transition Diagram DISTANCE_ANALYZER

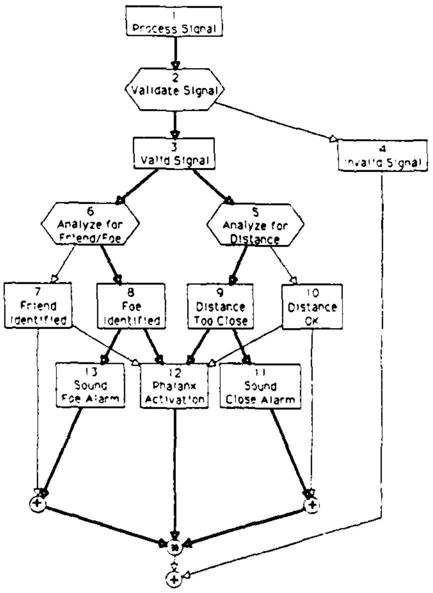


Figure 10 System Operations Diagram (Radar Signal Processing)

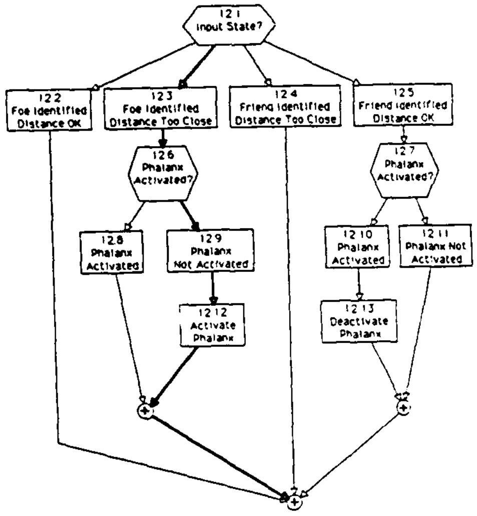


Figure 11 System Operations Diagram
(12 - Phalanx Activation)

```
task body DISTANCE_ANALYZER is
  SIGNAL_COPY   : SIGNAL_TYPE;
  DISTANCE      : MILES_TYPE;
  TOO_CLOSE     : BOOLEAN;
  SAFE_DISTANCE : MILES_TYPE := 200.0;
  procedure SOUND_CLOSE_ALARM is
  begin
    PUT_LINE ("Close alarm sounded");
  end SOUND_CLOSE_ALARM;
begin
  loop
    select
      accept ANALYZE_SIGNAL (SIGNAL : SIGNAL_TYPE) do
        SIGNAL_COPY := SIGNAL;
      end ANALYZE;
      -- determine distance:
      DISTANCE := SIGNAL_COPY.MILES;
      if DISTANCE < SAFE_DISTANCE then
        SOUND_CLOSE_ALARM;
        TOO_CLOSE := TRUE;
      else
        TOO_CLOSE := FALSE;
      end if;
      accept REPORT (CLOSE_STATUS : out BOOLEAN) do
        CLOSE_STATUS := TOO_CLOSE;
      end REPORT;
    or
      accept CHANGE_SAFE_DISTANCE (NEW_VALUE : MILES_TYPE) do
        SAFE_DISTANCE := NEW_VALUE;
      end CHANGE_SAFE_DISTANCE;
    end select;
  end loop;
end DISTANCE_ANALYZER;
```

Figure 12 Ada Code Fragment

Richard F. Vidale

Dr. Vidale has been a member of the faculty at Boston University since September, 1964, serving as Chairman of the Department of Electrical, Computer, and Systems Engineering from 1971 to 1981. He has been involved with applications of structured programming and software engineering since 1977, introducing courses at Boston University in software engineering and Ada in the early 1980's. His research includes: assessing Ada for typical command and control functions, the use of timing diagrams and associated design tools, extending Ada design methodologies, and Ada performance issues. Dr. Vidale has served as a consultant on software engineering and Ada to Chas. T. Main International, GTE, Naval Coastal Systems Center, The MITRE Corporation, Data General Corporation, The Charles Stark Draper Laboratory, Kollsman, Inc., and The Analytic Sciences Corporation.

Bard S. Crawford

Dr. Crawford received the B.S. and M.S. degrees in Aeronautical Engineering and the ScD degree in Aeronautics and Astronautics, all from the Massachusetts Institute of Technology, Cambridge. He is currently Manager of the Office of Ada Technology at The Analytic Sciences Corporation (TASC). He has been with TASC since 1968, where he has directed numerous programs related to the navigation, guidance, and control of missiles, aircraft, and the Space Shuttle. He has published a number of papers in navigation, estimation, and control and contributed to the book "Applied Optimal Estimation" edited by A. Gelb. He was previously employed at the Charles Stark Draper Laboratory, where he contributed to the development of the Apollo guidance and navigation system. His current research interests include the use of the Ada language in real-time concurrent systems and Ada Programming Support Environments.

SEDL — An Ada-Based Specification and Design Language

Gerry Fisher and Ann E. Kelley Sobel

IBM T.J. Watson Research Center
Yorktown Heights, New York 10598

Abstract

In creating a software product, it is essential to identify precisely and specify its functional characteristics. The use of formal notations to outline these characteristics allows the programmer to be concise and unambiguous, to support formal reasoning about the functional specification, and to provide a basis for verification of the resulting software. Given these advantages, a programmer would benefit from the use of a programming language that permits the integration of these formal functional characteristics (specifications) with the design and execution of the software. A coherent language that covers the transition from problem specification to implementation is referred to as a wide spectrum language [1]. The presented wide spectrum language SEDL, Software Engineering Design Language, combines specification and design into one language that is executable. SEDL provides the programmer with the potential to record and maintain all these views of a software product in an integrated and cohesive presentation.

I. Introduction

A wide spectrum Ada* -based programming language, SEDL [4], is presented that supports specification, design, and execution of software units. The availability of all three types of constructs in one language allows the potential of recording and maintaining all three views of a software product in an integrated and cohesive presentation. Incorporated within the language SEDL is a design methodology based upon the use of abstraction for specification and the process of refinement for incrementally adding program design details. Each refinement step adds both a design structure and additional lower level specifications which are included as components within that design structure. When applied repetitively, this results in a hierarchy of intermediate abstractions being recorded and presented to the reader. These specifications may be recorded with varying levels of formality. When completed, they form the basis for repeating the refinement step. This refinement process is terminated when implementation in Ada is straightforward.

In SEDL, one can use text for recording high-level specifications and designs less formally. The non-text components of the SEDL specifications can be checked for internal consistency by analyzers and compilers. This permits the programmer to trade off precision and expressiveness in design recording while maintaining the benefits of consistency checking. Using a design language that incorporates specifications provides certain freedoms:

- One can use text to specify the behavior or structure of the program informally. Later, one can refine that text into actual code, a more detailed textual specification, or some combination of the two.
- One can use set-theoretic notations to describe data structures before deciding on the representation of those data structures.
- One can specify the behavior of the software product using high level notations that can be compiled for the purpose of rapid prototyping. Later, one can refine those specifications into algorithms that are designed with the appropriate level of efficiency.

The identification of an implementation and code packaging strategy is an important component of SEDL. Without such a strategy, the programmer is exposed to the following difficulties.

- Different implementation strategies will be chosen, causing a fragmentation of the common base
- The code will lose some of its traceability back to the design
- Uncertainty will be introduced into the software development process at a point where it is not expected and therefore cannot be easily addressed

* Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

SEDL supports a methodology which allows the programmer to create and present abstract specifications that suppress implementation details. This methodology supports behavior and data modelling in conjunction with set-theoretic abstract types. Both behavior and data modelling are intended to separate the concerns associated with the use from those associated with the design of a software product. This separation enforces a level of modularity by separating the design decision (***how*** the software accomplishes the specification) from the specification of ***what*** the unit is intended to accomplish. In the following sections, we will present these concepts of behavior and data modelling along with a short discussion on the use of SEDL.

II. Behavior Modelling in SEDL

SEDL supports the principle of abstraction using behavior modelling. Abstraction permits separation of the conceptual aspects of a system from the implementation details. During software design, abstraction allows the designer to postpone structural and algorithmic considerations until the functional characteristics and data have been established. Therefore, design becomes the process of proceeding from abstract considerations to concrete representations.

Behavior modelling allows one to abstract the details of procedural logic in terms of a behavior specification, a special form of statement enclosed in < • and • >, which states the relationship of the program state variable values before a given statement or subprogram is executed to the values they may have as a result of executing the statement or subprogram. These behavior specifications are defined using a state transition model [2]. Given the set of states of the system Σ a transition function f is defined as follows:

$$f(X) = Y$$

> where X are the values of the variables in the system state σ and Y are their values in the subsequent system state σ'.

A nondeterministic behavior specification contains the keyword **any** and is interpreted as the arbitrary selection of an element from the specified set. If nondeterminism is introduced, the corresponding behavior specification must be defined as a transition relation where Y becomes a *set* of possible state variable values for the corresponding *set* of possible subsequent system states $\sigma'_1, \ldots, \sigma'_n$.

SEDL provides two predefined procedures, **Identity** and **Undefined**. The procedure **Undefined** is used to explicitly indicate that the specified behavior is undefined. The **Identity** procedure indicates that the behavior specified is the identity relation for the given values of the program state variables.

We will illustrate how one would write a SEDL specification of the program Merge. Using English, the function of Merge is to return a vector that is an ordered arrangement of the catenation of two given ordered vectors. The specification of Merge is as follows.

```
type Vector is array(Integer range < > ) of Integer;

function Is_Ordered (X: Vector) return Boolean
   < • for all I, J in X'RANGE : I < J ⇛ X(I) < = X(J) • > ;

function Is_Arrangement_Of (X,Y: Vector) return Boolean
   < • X'LENGTH=Y'LENGTH and (for all I in X'RANGE :
        Card({K in X'RANGE : X(K) = X(I)}) =
            Card({K in Y'RANGE : Y(K) = X(I)})) • > ;

function Merge (X,Y: Vector) return Vector
   < •Is_Ordered(X) and Is_Ordered(Y) → any Z in Vector :
        Is_Ordered(Z) and Is_Arrangement_Of(Z,X&Y) • > ;
```

The use of constructive, mathematically founded notation for behavior specification makes it possible to execute these SEDL specifications in order to perform rapid prototyping. A programmer would then be able to experiment with the functional behavior of a design without having to spend time developing the details of its implementation.

SEDL supports the concept of stepwise refinement which allows the programmer to incrementally add software design details from the decomposition of high-level specifications. Each refinement step adds both a design structure and additional lower-level specifications. The incremental addition of detail at each step of the refinement process postpones design decisions as long as possible and allows the designer to argue convincingly that the resulting software product is consistent with the design specifications. SEDL also supports the notion of design executability so that one could check the consistency between the specification of the design and the subsequent specification of its refinement. This check would ensure that the following relationship is true given that the relation f is a behavior specification and the relation g is the behavior of its refinement.

$$< * f * > \text{ is } g \;\equiv\; f \subseteq g$$

where

$$f \subseteq g \overset{\text{def}}{=} \forall \bar{X} \in \mathbf{D}(f)[\bar{X} \in \mathbf{D}(g)_{\mid \mathbf{D}(f)} \wedge f(\bar{X}) = g(\bar{X})]$$

$\mathbf{D}(f)$ denotes the domain of f and $\mathbf{D}(g)_{\mid \mathbf{D}(f)}$ denotes the domain of g restricted to the domain of f.

The specification of the function Merge given above is transformed into a SEDL design by refining the behavior specification of Merge into a program statement which can also contain additional behavior specification statements to be refined. This refinement is presented by following the original behavior specification with the keyword is. Each refinement step should reflect a definite design decision as to how a specified behavior is to be implemented. Both data and control structures are refined in this manner until the entire specification has been rewritten into code.

```
function Merge(X,Y: Vector) return Vector
  < * Is__Ordered(X) and Is__Ordered(Y) →
     any Z in Vector :Is__Ordered(Z) and Is__Arrangement__Of(Z,X&Y) *  > is
        Z :  Vector(1 .. X'LENGTH + Y'LENGTH);
        XF: constant Integer: = X'FIRST;
        YF: constant Integer: = Y'FIRST;
  begin
        < * X'LENGTH = 0  →  Z: = Y
           | Y'LENGTH = 0 → Z: = X
           | X(XF) < = Y(YF) → Z: = X(XF) and Merge(X(XF + 1 .. X'LAST), Y)
        else Z: = Y(YF) and Merge(X, Y(YF + 1 .. Y'LAST)) *  > ;
        return Z;
  end Merge;
```

Subsequent additional refinements will transform this SEDL design of Merge into sufficient detail so that efficient implementation in Ada is straightforward. The coding phase in SEDL should be viewed as merely adding a lowest level of refinement to the completed design. The implementation component of SEDL (Ada) checks the package specification, which includes information concerning its interface, for consistency when coupled with other independently-developed and compiled packages. Advantages to this approach include the protection of data, the promotion of independent testing, and the potentially substantial savings of compilation time.

III. Data Modelling in SEDL

Data modelling supports a major jump in abstraction since it is no longer necessary for the system designer to map the problem domain to the predefined data and control structures present in the implementation language (introduced by Hoare [3]). The designer may create abstract data types and functional abstractions and map the real-world domain to these programmer-created abstractions. SEDL also supports, through the use of data modelling, data encapsulation which defines a data structure by the operations performed on it. Since the data structure is packaged with its access routines in a single module, it is manipulated only by the contained access routines. Routines that use the data structure are not privy to the details of its data representation or manipulation.

Data modelling allows one to abstract the details of data representations in terms of SEDL abstract types and objects which include the set-theoretic abstract types such as sets, maps, relations and sequences. An abstract type consists of a collection of definitions of types and operations on objects of the defined type. A SEDL abstract type is defined in two parts: the specification part containing the model type declarations and behavior specifications for the operations of the type; and the body part containing the representation of the abstract type and the implementation of its operations. The model of the abstract type is the view of the type seen within the specifications of its operations. The constraint for the model states a condition that must always hold on the objects and the initialization specifies the initial value for the objects of this type.

As an example of data modelling in SEDL, let us consider the abstract data type, Smallset, consisting of integer sets whose cardinality does not exceed a given maximum size. This example uses data modelling to specify the abstract type while permitting the designer to select a data representation during another phase of development.

```
abstract type Smallset is                    —Assume Max__Size is a non negative Integer constant

     model

          type Smallset is set of Integer;

          constraint

               for all S in Smallset : Card(S) < = Max__Size;

          initial

               Empty;

     end model;

     function Query(I: in Integer; S: in Smallset) return Boolean

          < * I in S * > ;

     procedure Insert(I: in Integer; S: in out Smallset)

          < * Query(I,S) → Identity

               | Card(S) < Max__Size → S: = S + {I} * > ;

     procedure Remove(I: in Integer; S: in out Smallset)

          < * Query(I,S) → S:= S − {I} else Undefined * > ;

end Smallset;
```

In SEDL, the abstract type body may contain a representation or implementation of the abstract data type. The representation is typically a low-level type, such as an array or linked list, that permits an efficient implementation of the abstract type operations. The behavior of the operations of the abstract type must be respecified in terms of the operations of the representation type. Just as the model defines the type as seen in the specification of the operations, the representation defines the type as seen in the implementation of the operations. One must demonstrate that the restated behavior maps to the behavior of the model using the function ϕ that maps the representation of the data to the model data. Given a data item Z of the representation type T_R, $\phi(Z)$ must be a data item of the abstract type T_A. Using this function ϕ, we can demonstrate the correspondence between a function f_A defined on T_A and the function f_R defined on T_R. Pictorially, the following relation must hold to ensure the correctness of our data representation.

$$\begin{tikzcd}
T_A \arrow[r, "f_A"] & T_A \\
T_R \arrow[u, "\phi"] \arrow[r, "f_R"] & T_R \arrow[u, "\phi"']
\end{tikzcd}$$

$$f_A \circ \phi(Z) = \phi \circ f_R(Z)$$

Returning to our example, the type Smallset can be represented in SEDL by a record containing the elements in an array, Elem__Vector, and a count, Size, of the actual number of elements present in the set.

```
abstract type body Smallset is

    representation

        type Elem__Vector is array(1 .. Max__Size) of Integer;

        type Smallset is

            record

                Size: Natural range 0 .. Max__Size: = 0;

                V: Elem__Vector;

            end record;

        constraint            — No duplicates in a Smallset

            for all S in Smallset and J, K in 1 .. S.Size:

              J ≠ K ⇏ S.V(J) ≠ S.V(K);

        mapping of S in Smallset'representation to Smallset'model

            < * {S.V(I):I in 1 .. S.Size} * > ;

    end representation;


    function Query(I: in Integer; S: in Smallset) return Boolean

        < * exists J in 1 .. S.Size : S.V(J) = I * >

    is separate;

    procedure Insert(I:in Integer; S: in out Smallset)

        < * Query(I,S) → Identity

          | S.Size < Max__Size → S.Size, S.V(S.Size + 1): = S.Size + 1, I * >

    is separate;
```

```
        procedure Remove(I: in Integer; S: in out Smallset)
                < • exists J in 1 .. S.Size : S.V(J) = I → S.Size, S.V(J): = S.Size − 1, S.V(S.Size) • >
        is separate;
    end Smallset;
```

The behavior of the operations of Smallset are restated in terms of the chosen data representation. Using our representation of Smallset, one must show that for each S in Smallset and each I in **Integer** it is the case that I in ϕ (S) if and only if there exists a J in 1 .. S.Size such that S.V(J)=I. To demonstrate the correspondence of the two behaviors of Insert(I,S), one must show that for each object S of type Smallset and I in **Integer:** S.Size lt Max__Size if and only if Card(M(S)) < Max__Size; and if S.Size < Max__Size and not Query(I,S) then $\phi(S') = \phi(S) + \}I\{$ given that S′ is the new value of S produced by Insert(I,S).

IV. Summary

An Ada-based specification and design language, SEDL, is proposed that supports the development, recording, analysis, and review of software products. SEDL can be used to express all stages of the program development process. This wide spectrum language incorporates specification, design, and implementation using a methodology that is based upon the formal notations of set theory and logic in conjunction with software engineering principles. Behavior and data modelling separate the design decisions from the specification of the software unit. The implementation component of SEDL (Ada) prov:des the coupling of independently-developed packages and the safety of this assembly process. SEDL specifications and designs are executable and may therefore be used for the rapid prototyping of software products.

V. References

1. Bauer, F.L., M. Broy, G. Gnatz, W. Hesse, B. Krieg-Brückner, H. Partsch, P. Pepper, and H. Wöossner. Towards a Wide Spectrum Language to Support Program Specification and Program Development, *SIGPLAN Notices*, Vol. **13**, No. **12**, December, 1978, pp. 15-24.

2. Ferrentino, A.B. and H.D. Mills. State Machines and Their Semantics in Software Engineering, *IEEE Computer*, November, 1977, pp. 242-251.

3. Hoare, C.A.R. Proofs of Correctness of Data Representation. *ACTA Informatica*, Vol. **1**, 1972, pp. 271-281.

4. Software Engineering Design Language, Version 1.1, Language Reference Manual, IBM, 1987.

5. Shaw, M., G.T. Almes, J.M. Newcomer, B.K. Reid, and W.A. Wulf. A Comparison of Programming Languages for Software Engineering, *Comparing and Assessing Programming Languages Ada, C, Pascal*, A.R. Feuer and N. Gehani Editors, Prentice-Hall, 1984.

6. Wiener, R. and R. Sincovec. *Software Engineering with Modula-2 and Ada*, John Wiley & Sons, Inc., 1984.

Gerry Fisher is Manager, Specification and Design Languages, in the Computer Science Department at the IBM Thomas J. Watson Research Center. His group is involved in the development of SEDL, a program specification and design Language.

He has a Ph. D. in Mathematics from the University of Connecticut. He has taught Computer Science at various Universities, including the University of North Carolina at Chappel Hill, the Illinois Institute of Technology, and Clarkson College of Technology. He has worked for Burroughs, Digital Equipment Corporation, and TeleSoft on programming language design and implementation.

In 1979 he joined the Courant Institute at New York University as a Research Associate Professor to work on the newly developed language, Ada. He directed the NYU Ada Project until August 1982. While at NYU he founded the Ada Implementor's Group which evolved into AdaTEC and now SIGAda, the ACM Special Interest Group on the Ada Language. He was the first Chairperson of AdaTEC and of SIGAda. Dr. Fisher was an Ada Distinguished Reviewer for the US Department of Defense and a member of the Ada Board. He presently serves on the ISO Ada Language Maintenance Committee.

Ann E. Kelley Sobel recieved her B.S. in Statistics from the University of Akron in 1980, a B.S. in Mathematics from Akron University in 1981, her M.S. in Computer Science from the Ohio State University in 1982, and her Ph.D. in Computer Science from Ohio State in 1986.

Dr. Sobel is a member of the Research Staff at the IBM Thomas J. Watson Research Center. Her background includes work in specification and design languages, and verification of concurrent programming languages.

She has been a member of Sigma Xi since 1985.

An Algebra for Real-Time Process Decomposition

Mike Adler

Control Data Corporation
Corporate Research and Engineering

Abstract

This paper describes a formal, top-down method that takes a transformation schema process (used for describing real-time processes) and logically implements it as a network of smaller data transformations and control transformations, based on its input and output data flows and control flows. This work generalizes the author's previous work by extending it from data flow diagrams (structured analysis) to transformation schemas (real -time analysis), reducing the number of operators in the algebra from seven to one, adding pre- and post-conditions, removing limitations due to the previous notation, and producing a result that is suitable for automatic layout. The control flow notation of transformation schemas is extended to include both signals and predicates. Predicates are conditions defined on data flows and have the same control properties as signals. Criteria are given for when to decompose a process, when to decompose a flow, and when to terminate the decomposition process.

I. Introduction

Transformation schemas are used for describing real-time processes [3]. This paper describes a formal, top-down method that takes a transformation schema process and logically implements it as a network of smaller data transformations and control transformations, based on its input and output data flows and control flows. A data transformation can be further decomposed as a transformation schema process. A control transformation may be decomposed into a network of smaller control transformations, or into a state-transition diagram.

This work generalizes the author's previous work [1] and extends it from data flow diagrams (structured analysis [2]) to transformation schemas (real-time analysis [3]). The previous work is generalized by eliminating the matrix-based part of the algebra and making it completely graph-based (although the matrix is retained in the specification for ease of use). This reduces the number of operators in the algebra from seven to one, adds pre- and post-conditions, removes limitations due to the previous notation, and produces a result that is suitable for automatic layout.

The control flow notation in [3] is extended to include both signals and predicates. Predicates are conditions defined on data flows (similar to the limited form predicates of [4]) and have the same control properties as signals. Specifications are extended to support input and output control definitions and control references in the transform matrix. Only flow connections are specified. The algebra generates the decomposition data and control transformations (and their local flow inter-connections) based on how the input and output flows group together.

A decomposition is produced in stages. The first stage uses the new algebra to produce a data flow diagram from the data flows in the specification. A single control transformation is defined. Signals are allocated to the control transformation, with connection through to data transformations as needed to meet the specification. Predicates connect data transformations to the control transformation.

The second stage uses the new algebra to produce a control flow diagram (a network of smaller control transformations) from the control flows of the single control transformation. This step is completely symmetric to the way that the data flow diagram is produced (but may be skipped if the number of control flows is small). In the last stage, each un-decomposed control transformation is decomposed to a state-transition diagram. This is done using a different technique that requires augmenting the specification at this point in the decomposition process. Again, only control flows are specified. The states and transitions are generated from the augmented specification.

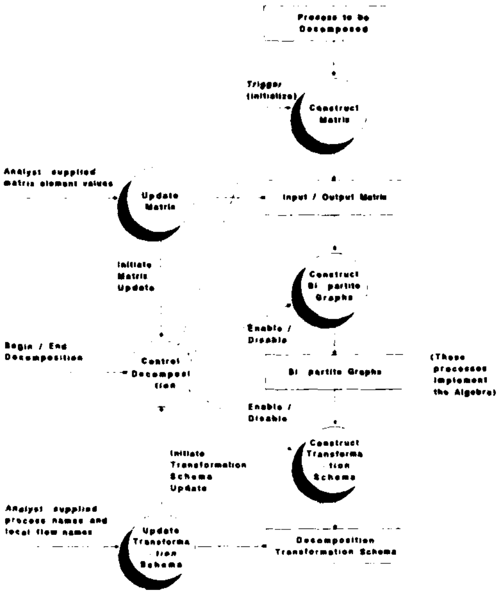
The decomposition process continues through further decomposition of the data transformations. Criteria are given for when to decompose a process, when to decompose a flow, and when to terminate the decomposition process.

The objective of this algebra is to provide a rigorous technique for real-time process decomposition at the Ward and Mellor [5] level. A grammar [6] based on directed acyclic graph (DAG) [7] generation from a bi-partite (two partition) graph is defined and presented in the style of axiomatic programming [8], where the nodes defined by the grammar form a very simple algebra. As with the previous work [1], this is a DAG construction problem and is not the same as DAG node count reduction [9] or compiler code optimization using DAGs [10]. The acyclic nature of the decomposition means that it contains no feedback loops (which are handled outside of the algebra). The defined grammar is applied to an example problem [11] to find the first level decomposition of a robot controller.

This algebra addresses data flow and control flow. It does not address at least the following: algorithms, data composition, data or procedural constraints, design objects. It is expected that in the future, the algebra will be integrated with these capabilities in other techniques.

For this paper, a *transformation schema* consists of a *data flow diagram* (*DFD*), a *control flow diagram* (*CFD*) and a set of control flows that connect the data and control transformations. These follow the Ward and Mellor conventions. A *DFD* will consist of *data transformations* (solid circles) and *data stores* (solid parallel lines), with input, output and connecting *data flows* (solid arrows). A *CFD* will consist of *control transformations* (dashed circles) and *control stores* (dashed parallel lines), with input, output and connecting *control flows* (dashed arrows). The control flows that connect the data and control transformations are also represented by dashed arrows. Figure 1 is an example of a transformation schema that shows how the algebra fits into the transformation schema process decomposition method.

The paper is organized into the following sections: grammar, interpretations, operator, decomposition, state-transition overview, tools and outlook. The grammar section discusses the graph basis of the grammar, the definitions of the grammar that derive from this, and the invariants of grammatical expressions. The interpretations section discusses the two ways of interpreting the sentences of the grammar -- as a matrix, and as a graph. The operator section discusses the motivation for the grammar, the operator definition, and the pre- and post-condition constraints that are put on the operator. The decomposition section discusses the overall decomposition process, from initiation to termination, distinguishes the data and control components, and defines a decomposition quality metric. The state-transition diagram section overviews an extension to the algebra that is the next logical transformation step. The tools section discusses a prototype of an automated algebra tool and how the algebra might be built into current software engineering tools for use in a production environment. Outlook for further transformations is in the final section.



Process to be Decomposed
Trigger (Initialize)
Construct Matrix
Analyst supplied matrix element values
Update Matrix
Input / Output Matrix
Initiate Matrix Update
Construct Bi partite Graphs
Enable / Disable
Begin / End Decomposition
Control Decomposition
Bi partite Graphs
(These processes implement the Algebra)
Enable / Disable
Initiate Transformation Schema Update
Construct Transformation Schema
Analyst supplied process names and local flow names
Update Transformation Schema
Decomposition Transformation Schema

Figure 1. The Algebra as Part of the Process Decomposition Method

II. Grammar

The graph basis of the grammar is presented first, followed by definitions of the symbols, terms and sentences of the grammar. Canonical forms that are used in the interpretations, operator, and pre- and post-condition definitions are derived. Appendix A discusses the motivation for a new algebra relative to the original algebra presented in [1].

A. Graph Basis for the Grammar

Consider DAGs of the form in Figure 2. This graph is connected and is called a *context graph*. Its vertex-connectivity is 1 and the context vertex is its one and only cut-vertex. The input, context and output vertices are all distinct. There is no directed path between any pair of input vertices or between any pair of output vertices. There is a directed path from each input vertex, through the context vertex, to each output vertex. This graph is equivalent to the complete bi-partite graph of Figure 3.

The problem to be solved is the construction of a DAG to replace the context vertex. The connectivity of the input and output vertices is reduced to that specified by a matrix, called the *input-output matrix* (*I/O matrix*). This matrix is equivalent to the adjacency matrix of a bi-partite graph which is connected but not complete (one with reduced connectivity relative to Figure 2). Appendix B discusses the equivalence of the I/O matrix and the adjacency matrix.

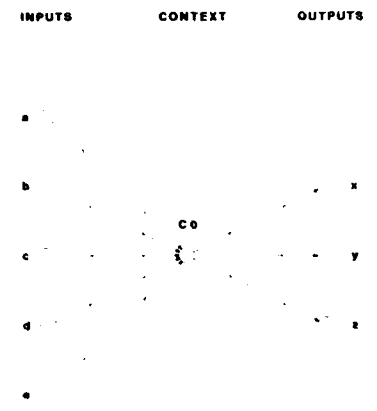


Figure 2. Context Graph General Form (and Example)

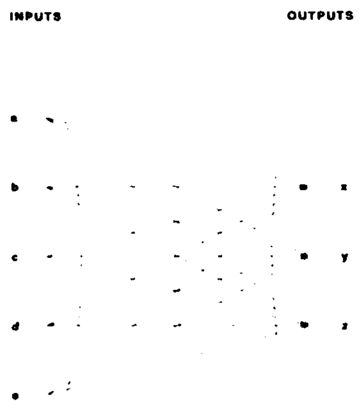
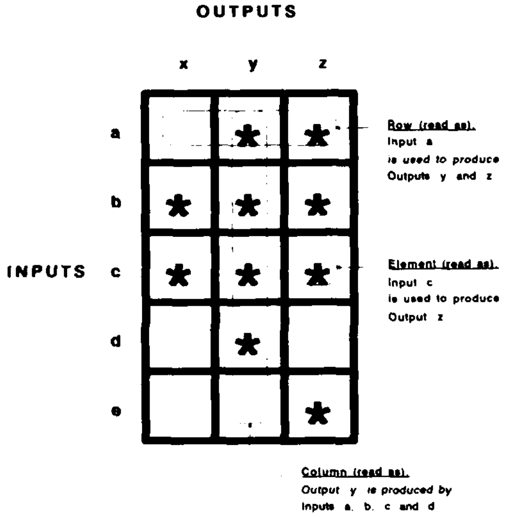


Figure 3 Complete Bi-partite Graph Corresponding to Figure 2

The constructed DAG together with its input and output vertices is called a *decomposition graph*. The DAG itself is called the *decomposition of the context*.

As an example, suppose the connectivity of the context graph in Figure 2 is to be reduced according to the connectivity specified (marked) in the I/O matrix of Figure 4. By applying the algebra to the bi-partite graph in Figure 5 (corresponding to the I/O matrix of Figure 4), the context node (C0) of Figure 2 is replaced by a DAG (C1-C2-C3-C4) to produce the decomposition graph in Figure 6. This graph has directed paths only between the inputs and outputs that were specified by the analyst in the I/O matrix of Figure 4 (corresponding to the bi-partite graph in Figure 5).



		OUTPUTS		
		x	y	z
INPUTS	a		*	*
	b	*	*	*
	c	*	*	*
	d		*	
	e			*

Row (read as):
Input a
is used to produce
Outputs y and z

Element (read as):
Input c
is used to produce
Output z

Column (read as):
Output y is produced by
Inputs a, b, c and d

Figure 4. I/O Matrix for Context Graph Decomposition Specification

In practice, the marked binary elements of the matrix specify that a particular input data or control flow is used (either conditionally or unconditionally) in the production of a particular output data or control flow. The selected elements of the matrix are a subset of the context and define a relation on the inputs and outputs called: *is used to produce*. For example, in the matrix of Figure 4, row c, column z is marked, so: input c *is used to produce* output z.

B. Symbols of the Grammar

The sentences of the algebra are made up of three classes of *symbols*: labels, nodes, and partitions.

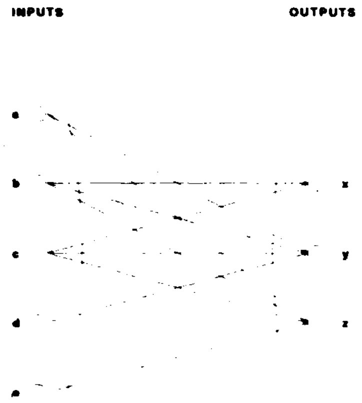


Figure 5. Bi-partite Graph Corresponding to Specification Matrix Figure 4.

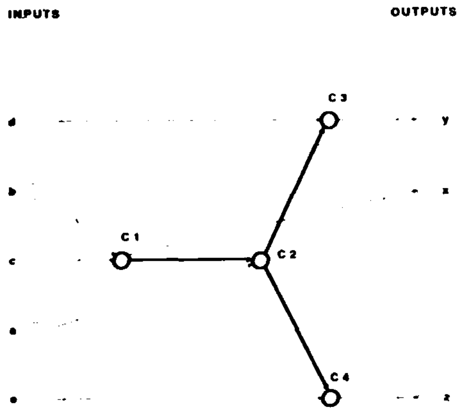


Figure 6. Decomposition DAG (C1 - C2 - C3 - C4) Replaces Context (C0) (see Figure 2)

Labels. *Labels* are denoted by <u>lower case bold letters</u>:

> The *input labels* (beginning alphabet letters): **a b c ...**

> The *output labels* (ending alphabet letters): ... **x y z**

Labels correspond to stores and flows. The input and output label sets are disjoint. In practice, this means that when a process has an output flow to a store and an input flow from the same store, the flows to and from the store are treated as distinct by the algebra, even though they have the same name. Non -store input and output flows of the same name are treated as representing different flow iterations, or as before-and-after flow updates, or as being otherwise distinct.

invariant: Each input label and each output label of the graph appears once.

Nodes. *Nodes* are denoted by <u>upper case letters</u>:

> The *input nodes* (beginning alphabet letters): A B C ...

> The *output nodes* (ending alphabet letters): ... X Y Z

> The *intermediate nodes* (middle alphabet): ... L M N ...

Nodes correspond to processes. Nodes are structured, consisting of four sets: input links, input labels, output labels and output links. The indegree of a node is ¦{input links}¦ and the outdegree is ¦{output links}¦. A node that has one input label, no output labels, indegree = 0, and outdegree > 0 is an input node (Figure 7a). Similarly, a node that has one output label, no input labels, outdegree = 0, and indegree > 0 is an output node (Figure 7b). All other nodes are intermediate nodes. Input and output nodes are the minimum cases of distributor and collector nodes -- having one or more input or output labels, respectively (Figure 7c/d). Distributor and collector nodes are important in defining data and control flow diagrams.

Node links are denoted by node names. Two nodes, M and N, are adjacent if the name of M appears as an input link of N and the name of N appears as an output link of M. M is said to be to the left of N, and N is said to be to the right of M (Figure 8). This may be stated by the adjacent predicate:

> adjacent (M N) [M is on the left and N is on the right]

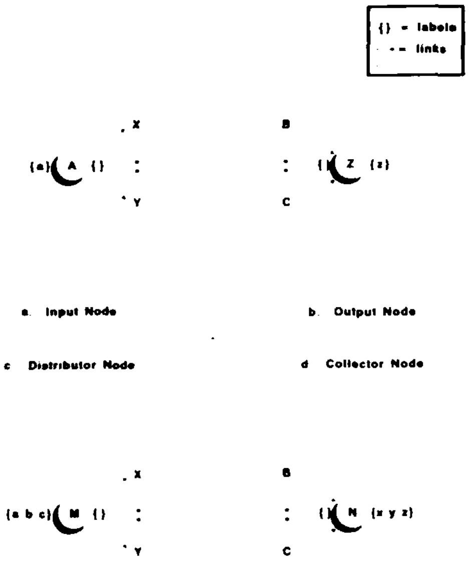
invariant: The nodes of the graph form a directed acyclic graph (DAG).

Links may also be characterized as primary or secondary, with definitions from [1]:

> A link connecting two nodes for which there is not a longer directed path is a *primary link*,

A link connecting two nodes for which a longer directed path exists is a *secondary link*.

invariant: All graph links are primary.



a. Input Node

b. Output Node

c Distributor Node

d Collector Node

Figure 7 Graph Interpretation of Distributor / Collector Nodes
(input / output is the minimum case - one label)

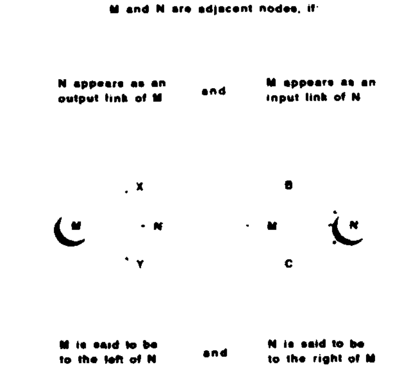

M and N are adjacent nodes, if:
N appears as an output link of M
and
M appears as an input link of N
M
X
N
Y
B
M
C
N
M is said to be to the left of N
and
N is said to be to the right of M

Figure 8 Adjacent Nodes

Partitions. *Partitions* are denoted by an upper case P followed by an identifier:

The *input partition*:
PI: upper case I

The *output partition*:
PO: upper case O

The *intermediate partitions*:
Pa: where a is a lower case letter:
a b c ... x y z

A partition is a set of non-adjacent nodes. There is one input partition, PI, and one output partition, PO. The input partition is the left-most partition (its nodes have no input links). The output partition is the right-most partition (its nodes have no output links). An intermediate partition is to the right of the input partition and to the left of the output partition.

Two partitions are either ordered or independent. Two partitions are ordered if they are adjacent, or if they are connected by a chain of adjacent partitions (the right partition is reachable from the left partition). Two partitions are adjacent if they contain at least one pair of nodes that are adjacent. As with nodes, this may be stated by the adjacent predicate:

adjacent (Pa Pz) [Pa is on the left and Pz is on the right]

All adjacent node pairs in adjacent partitions have the same orientation (left to right) -- this is called *uniform* orientation and the orientation of two adjacent partitions is said to be the same as the uniform orientations of their adjacent nodes. Two partitions are independent if they are not adjacent and if there is no chain of partitions connecting them.

The initial graph constructed from the I/O matrix is bi-partite (has two partitions) -- it contains the input and output partitions and no intermediate partitions (Figure 5, for example). The final graph, after applying the algebra, will be multi-partite -- it will have two or more partitions (Figure 9, for example).

invariant:
The partitions of the graph form a multi-partite, uniform, directed acyclic graph (mu-DAG).

C. Terms of the Grammar

The *terms* of the algebra are nodes. The algebra operation will change the sentences of the algebra by operating on the terms of the sentences (its nodes). The terms of the algebra are contained in partitions.

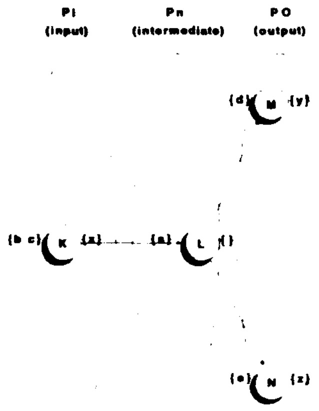


Figure 9. Multi-partite Graph with Nodes Replacing Vertices (see Figure 6)

D. Sentences of the Grammar

The *sentences* of the algebra are constructed from partitions according to the following rules:

A partition is a sentence.

A multi-partite, uniform, directed acyclic graph (mu-DAG) of partitions is a sentence.

A subgraph of a mu-DAG which is also a mu-DAG is a sentence.

E. Definition of Canonical Forms

Canonical forms for terms and sentences can now be defined. These forms will be used in the definition of sentence interpretations and the sentence operator.

Canonical Terms. A canonical term is a set of one or more nodes (within a partition) represented by an upper case italic letter.

Recall that a node has a structure consisting of four sets:

{input links}	adjacent node names (from the left)
{input labels}	input label symbols
{output labels}	output label symbols
{output links}	adjacent node names (to the right)

A set or a member of one of the sets of a term will be denoted by a short set name as a function applied to the term, so that:

inlabels (*T*) [plural form]

is the input label set of term *T*, and:

inlabel (a *T*) [singular form] (or just a, if it is unambiguous)

is one input label of term *T*. These forms apply to all of the nodes in the term. A term may be broken into subgroups and a single node of a term may be designated by the node function:

node (N *T*) (or just N, if it is unambiguous)

and the partition which contains a term may be designated by the partition function:

partition (*T*)

The input links of a node may belong to one or more partitions and may be grouped into one term per left adjacent partition. Similarly for output links and right adjacent partitions. The canonical form of a node then becomes:

{*K L* ...} {a b c ...} {... x y z} {... *M N*} <1>

Since a partition is a set of nodes and a set of nodes contained in a partition is a term, all partitions are *maximal* terms, with canonical form:

{*T*} <2>

or (expanding <2> to subgroup form),

{*K L M* ...} <3>

Canonical Sentences. A canonical sentence is a set of one or more sentences represented by an upper case bold letter.

The canonical form for a sentence is:

S <4>

or (expanding <4> to subgraph form),

P Q R ... <5>

Since the smallest sentence structure consists of a single partition, all partitions are *minimal* sentences.

The canonical form for a sentence, <5>, becomes:

[PI] [[...] Pn [...]] [PO] <6>

where: [] denotes optional, subject to the constraint that there is at least one partition present (actual partition adjacency is determined by node adjacency, not the apparent adjacency of canonical sentence form).

Partitions in a sentence, <6>, may be expanded to their canonical term forms, <2> or <3>, and terms of one node to their canonical node form, <1>.

III. Interpretations

The sentences of the algebra may be interpreted in two ways: as a matrix, and as a graph. The matrix interpretation constructs a matrix from a sentence. This is effectively the inverse of what the algebra operator does (construct a sentence from the bi-partite graph derived from a matrix). The matrix interpretation is used to compare the initial matrix to that which results from applying the operator. This comparison measures decomposition quality. The graph interpretation constructs a graph that corresponds to a decomposition final sentence. The interpretations set the relationships between an I/O matrix, a sentence, and a decomposition graph.

A. Matrix Interpretation of a Sentence

The *matrix interpretation* of a sentence is the set of matrix elements that the sentence generates by pairing inputs and outputs. This is used to define the *equivalence* of sentences:

> Two sentences which have the same matrix interpretation are *equivalent*.

Matrix Interpretation Procedure. Given a sentence, S, the I/O matrix corresponding to the sentence is constructed by the following procedure:

```
-   relabel the current input and output
    partitions as intermediate partitions
    (PI -> Pa; PO -> Pz);
-   create new, empty input and output
    partitions (PI, PO);
-   for each node of each intermediate
    partition of the sentence
    (S = P Q R ...
       = P Pn R ... = P {T} R ...
       = P {node (N T)} R ...):
    - for each input label of the node
      (inlabel (a N)):
      - create an input node in the input
        partition ({} {a} {} {});
      - create an output link from the
        new input node to the original
        node ({} {a} {} {N});
      - replace the input label in the
        original node with an input link
        from the new input node
        ({K L ... + A} {a b c ... - a}
        {... x y z} {... M N});
    - for each output label of the node
      (outlabel (z N)):
      - create an output node in the
        output partition ({} {} {z} {});
      - create an input link to the new
        output node from the original
        node ({N} {} {z} {});
      - replace the output label in the
        original node with an output link
        to the new output node
        ({K L ...} {a b c ...}
        {... x y z - z} {... M N + Z});
-   create the node adjacency matrix from
    this expanded sentence:
    - the source nodes include the
      intermediate nodes and the input
      nodes (output nodes need not be
      included, see Appendix B);
    - the destination nodes include the
      intermediate nodes and the output
      nodes (input nodes need not be
      included, see Appendix B);
-   from the adjacency matrix, compute
    the reachability matrix (transitive
    closure);
-   the subset of the reachability matrix
    that includes only the input and
    output nodes is the required I/O
    matrix -- in other words, the I/O
    matrix elements correspond to those
    input-output node pairs where the
    output nodes are reachable from the
    input nodes through the nodes of the
    sentence.
```

Matrix Interpretation Example. The example sentence following corresponds to Figure 9 which is the decomposition of the context in Figure 2, as specified by the I/O matrix in Figure 4:

```
S = PI Pn PO = {K} {L} {M N}
  = {{} {b c} {x} {L}}          K
    {{K} {a} {} {M N}}          L
    {({L} {d} {y} {})
     ({L} {e} {z} {})}          MN    <7>
```

Applying the matrix interpretation procedure to <7> first causes new input and output partitions to be generated and populated giving:

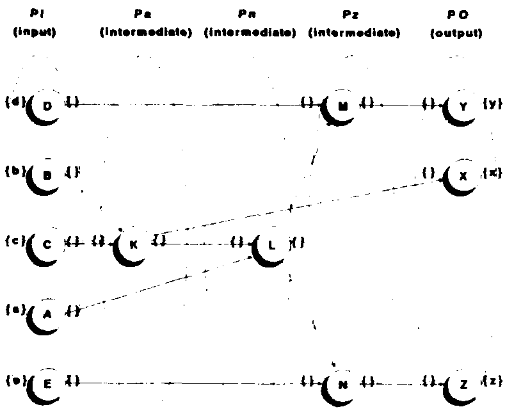
```
S = PI Pa Pn Pz PO
  = {A B C D E} {K} {L} {M N} {X Y Z}
  = {({} {a} {} {L})                 <8>
     ({} {b} {} {K})
     ({} {c} {} {K})
     ({} {d} {} {M})
     ({} {e} {} {N})}            ABCDE
    {{B C} {} {} {L X}}          K
    {{A K} {} {} {M N}}          L
    {({D L} {} {} {Y})
     ({E L} {} {} {Z})}          MN
    {({K} {} {x} {})
     ({M} {} {y} {})
     ({N} {} {z} {})}            XYZ
```

The only labels in <8> are in the input and output nodes (Figure 10). Continuing with the matrix interpretation procedure, the adjacency matrix of <8> is shown in Figure 11 and the reachability matrix is computed in Figure 12. The indicated subset of the reachability matrix is the required I/O matrix. Compare this subset to Figure 4.

B. Graph Interpretation of a Sentence

The *graph interpretation* of a sentence is a set of processes, each of which may have external input and output flows and which are inter-connected by a set of local flows. This is used to define the *congruence* of sentences:

> Two sentences which have the same graph interpretation are *congruent.*



Only input and output nodes have labels in this form of the graph

Each input and output node also has only a single link
(a result of the invariant that each label appear only once)

Figure 10. Multi-partite Graph with Labels Extracted to New PI / PO
(see Figure 9)

OUTPUTS

INPUTS

	K	L	M	N	X	Y	Z
K	0	1	0	0	1	0	0
L	0	0	1	1	0	0	0
M	0	0	0	0	0	1	0
N	0	0	0	0	0	0	1
A	0	1	0	0	0	0	0
B	1	0	0	0	0	0	0
C	1	0	0	0	0	0	0
D	0	0	1	0	0	0	0
E	0	0	0	1	0	0	0

decomposition graph nodes

input nodes

decomposition graph nodes

output nodes

Figure 11. Adjacency Matrix Corresponding to Figure 10.

OUTPUTS

INPUTS

	K	L	M	N	X	Y	Z
K	0	1	1	1	1	1	1
L	0	0	1	1	0	1	1
M	0	0	0	0	0	1	0
N	0	0	0	0	0	0	1
A	0	1	1	1	0	1	1
B	1	1	1	1	1	1	1
C	1	1	1	1	1	1	1
D	0	0	1	0	0	1	0
E	0	0	0	1	0	0	1

decomposition graph nodes

input nodes

decomposition graph nodes

output nodes

Compare to Figure 4 – this represents all input/output node pairs such that the output node is reachable from the input node through the decomposition graph nodes

Figure 12. Reachability Matrix Corresponding to Figure 11

Graph Interpretation Procedure. Given a sentence, S, the DFD or CFD component of a transformation schema corresponding to the sentence is constructed by the following procedure:

- each node of the sentence becomes a process;
- for each node of each partition of the sentence:
  - each input label becomes an external input flow or an input flow from a store (depending on the context);
  - each output label becomes an external output flow or an output flow to a store (depending on the context);
- for each adjacent node pair:
  - the corresponding processes are connected by a local flow from the process corresponding to the left node of the adjacent pair to the process corresponding to the right pair of the adjacent node pair;
- initial node layout of the graph is according to partition:
  - partition PI, the input partition, is the left-most partition;
  - partition PO, the output partition, is the right-most partition;
  - partitions Pn, the intermediate partitions, are placed between partitions PI and PO in partition adjacent order;
  - the nodes of each partition are arranged vertically so that local flow overlap is eliminated or minimized.

Graph Interpretation Example. The example sentence, <7>, repeated here for convenience, is the decomposition of the context in Figure 2, as specified by the I/O matrix in Figure 4:

```
S = PI Pn PO = {K} {L} {M N}
  = {{} {b c} {x} {L}}       K
    {{K} {a} {} {M N}}       L
    {({L} {d} {y} {})
     ({L} {e} {z} {})}       MN         <7>
```

Following the graph interpretation procedure:

- nodes K, L, M, and N become processes;
- labels become external flows:

process	input flows	output flows
K	b, c	x
L	a	-
M	d	y
N	e	z

- links become local flows:

from process	to process
K	L
L	M
L	N

This produces Figure 13. Compare this to Figure 9 (the node version).

C. Conjecture

Equivalent sentences are congruent, and conversely.

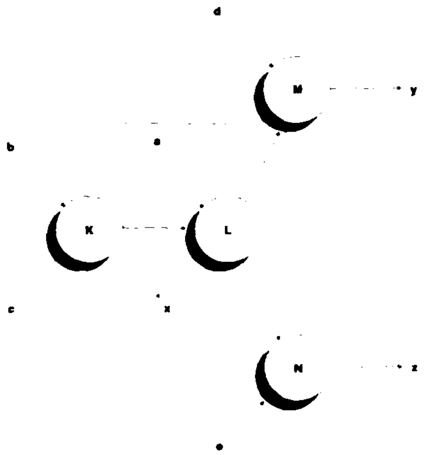


Figure 13. Graph Interpretation Example

IV. Operator

The algebra *operator* operates indirectly on the sentences of the algebra by operating directly on the nodes of a sentence's terms. An *initial sentence* is constructed from the bi-partite graph corresponding to an I/O matrix. The input partition contains input nodes and the output partition contains output nodes. The function of the operator is to combine input and output nodes into process nodes (possibly generating intermediate partitions) that will be the data or control transformations of the DFD or CFD component of the decomposition transformation schema. The final result of this combining process is the *terminal sentence*.

A. The Merge Operator

The node combining operation is called merge. The canonical form for the operator is:

node1.node2

where the connective period (.) means that the combined form represents a single node. Two nodes are merged by the following procedure:

- for each of the four sets (input links, input labels, output labels, and output links) of the two nodes being merged:
  - the set of the merged node is the *union of the corresponding sets of* the two nodes being merged.

so that if (Figure 14a):

```
M = {J K ...}    {a b ...}    {... x y}
    {... P Q}

N = {K L ...}    {b c ...}    {... y z}
    {... Q R}
```

then (Figure 14b):

```
M.N = {J K ...} union {K L ...}
      {a b ...} union {b c ...}
      {... x y} union {... y z}
      {... P Q} union {... Q R}

    = {J K L ...} {a b c ...}
      {... x y z} {... P Q R}
```

If the identity node, I, is defined as:

I = {} {} {} {}   [all null sets]

then the algebra can be derived immediately from the definitions of the merge operator (.) and the identity node (I), and from the properties of union, without further proof:

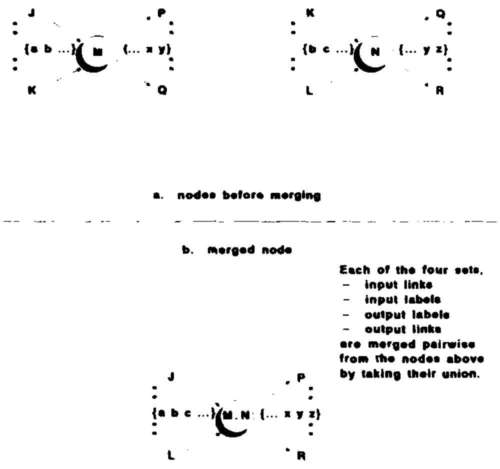
(1) I.I = I   identity

(2) N.I = I.N = N   identity

(3) N.N = N   idempotence

(4) M.N = N.M commutative

(5) L.(M.N) = (L.M).N associative



a. nodes before merging

b. merged node

Each of the four sets,
- input links
- input labels
- output labels
- output links

are merged pairwise from the nodes above by taking their union.

Figure 14. Merging Nodes

B. Termination

As defined above, the merge operator forms a simple and powerful algebra. In fact, it's powerful enough to reduce every bi-partite graph to a single node. This is a trivial decomposition since a single context node is the starting point. Some criteria are needed to define the subset of possible graphs that are acceptable as the DFDs and CFDs of a transformation schema. This is where the need for distributor and collector nodes arises.

Each input node in the input partition is a minimal distributor node. Each output node in the output partition is a minimal collector node. A process node is distinguished from a distributor or collector node in that it meets one or both of the following criteria:

the node has at least one input label and at least one output label,

the node has indegree > 0 and outdegree > 0.

The termination criteria for combining nodes is that the graph remain connected and either:

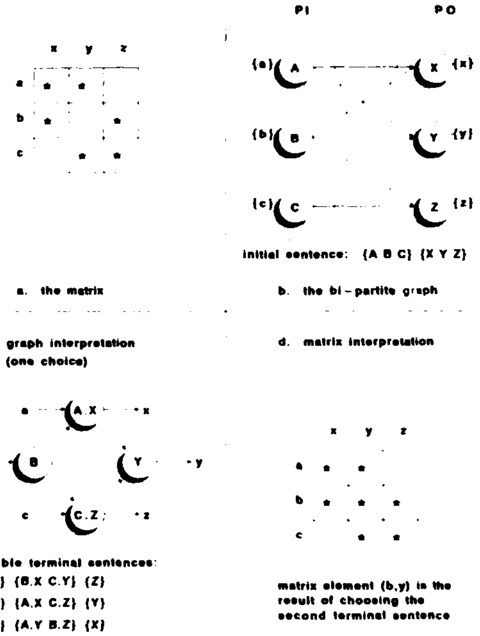
(*minimal*) distributor nodes are not adjacent to collector nodes, or

(*maximal*) all distributor and collector nodes have been converted to process nodes.

(Note: The termination condition, along with the invariants for partitions (mu-DAG) and links (primary), may force the final partition mu-DAG to be linearly ordered (a chain). This would mean that the final decomposition graph (DAG) is weakly ordered [12]. If this is the case, is the generality of the algebra reduced? This question requires further resolution.)

C. Pre- and Post-Conditions

In the case above, where the graph was reduced to a context node (before the introduction of the termination criteria), some links must have been added by some of the merge operations since the bi-partite graph corresponding to a context node is complete (all inputs are connected to all outputs). Merge operations that do not change the matrix interpretation of a sentence are called *conservative* and are preferable to the *non-conservative* operators which do. Non-conservative operators effectively change the specification (but make choices and resolve conflicts in doing so, see Figure 15). To implement preference for conservative selection, pre- and post-conditions are added to the merge operator.



initial sentence: {A B C} {X Y Z}

a. the matrix

b. the bi-partite graph

c. graph interpretation (one choice)

d. matrix interpretation

possible terminal sentences:
{A} {B.X C.Y} {Z}
{B} {A.X C.Z} {Y}
{C} {A.Y B.Z} {X}

matrix element (b,y) is the result of choosing the second terminal sentence

Figure 15. Making Choices

Pre- and post-condition notation is borrowed from the axiomatic programming notation for program precondition and result, [8]. The canonical form for merge with pre- and post-conditions is:

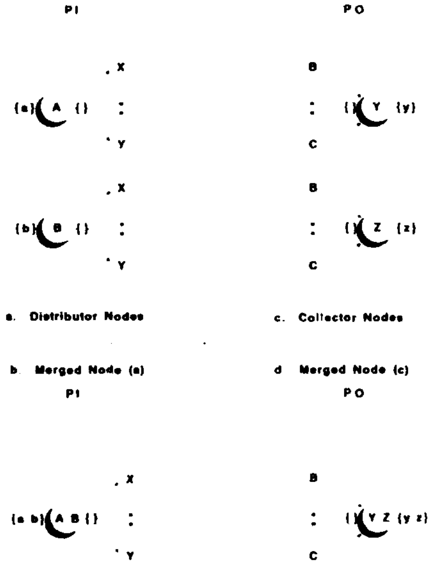
```
pre-conditions
{node1.node2}
post-conditions
```

There are four condition types:

```
node partition match
  (equal, adjacent)

node set* match
  (equal, subset, intersect)

*(inlink, inlabel, outlabel, outlink)

node type
  (distributor, collector, process)

node adjacency
  (adjacent, non-adjacent)
```

Conservative Merges. There are six conditions under which conservative merges occur between two nodes:

```
(1) Pre-conditions    (Figure 16a/b)
    partition(A)    = partition(B)
    type(A)         = distributor
    type(B)         = distributor
    outlinks(A)     = outlinks(B)

    Post-conditions
    partition(A.B) = partition(A)
                   = partition(B)
    outlinks(A.N)  = outlinks(A)
                   = outlinks(B)
```



a. Distributor Nodes

c. Collector Nodes

b Merged Node (a)

d Merged Node (c)

Figure 16 Conservative Merges (Distributor / Collector Nodes)

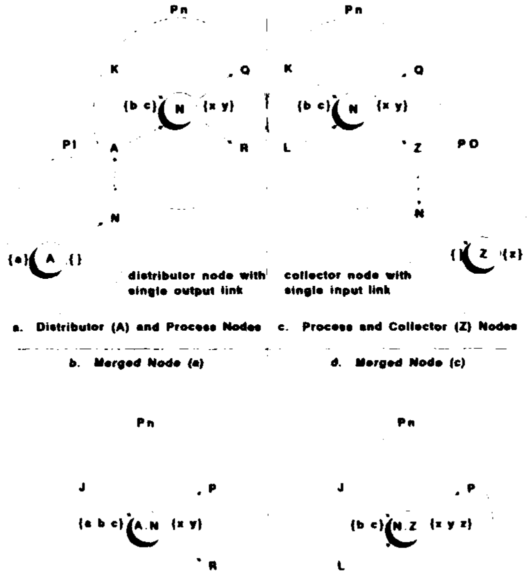
This says that two distributor nodes which are adjacent to the same nodes can be combined.

```
(2) Pre-conditions    (Figure 16c/d)
    partition(Y)    = partition(Z)
    type(Y)         = collector
    type(Z)         = collector
    inlinks(Y)      = inlinks(Z)

    Post-conditions
    partition(Y.Z) = partition(Y)
                   = partition(Z)
    inlinks(Y.Z)   = inlinks(Y)
                   = inlinks(Z)
```

This says that two collector nodes which are adjacent to the same nodes can be combined.

```
(3) Pre-conditions    (Figure 17a/b)
    adjacent(partition(A) partition(N))
    type(A)         = distributor
    type(N)         = process/collector
    outlinks(A)     = {N}

    Post-conditions
    partition(A.N) = partition(N)
```

This says that a distributor node attached to only one other node can be combined with it.



a. Distributor (A) and Process Nodes

c. Process and Collector (Z) Nodes

b. Merged Node (a)

d. Merged Node (c)

Figure 17 Conservative Merges (Distributor - Process / Process - Collector)

```
(4) Pre-conditions    (Figure 17c/d)
    adjacent(partition(N) partition(Z))
    type(N)         = process/distributor
    type(Z)         = collector
    inlinks(Z)      = {N}

    Post-conditions
    partition(N.Z) = partition(N)
```

This says that a collector node attached to only one other node can be combined with it.

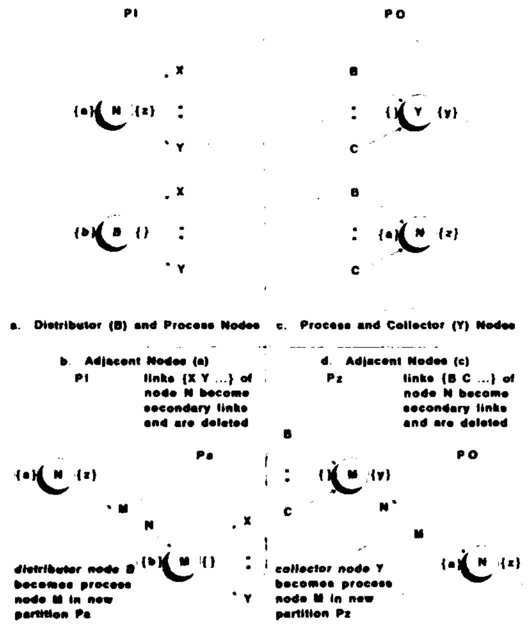
There are also two conservative null-merge conditions that convert distributor and collector nodes to process nodes through a coercive adjacency post-condition. The null-merge part may be thought of as:

```
pre-conditions {I.I} post-conditions
```

with the pre- and post-conditions specifying the actual nodes.

```
(5) Pre-conditions    (Figure 18a/b)
    partition(N)    = partition(B)
    type(N)         = process
    type(B)         = distributor
    outlinks(N)     = outlinks(B)

    Post-conditions
    adjacent(partition(N) partition(M))
    type(M)         = process (was B)
    adjacent(N M)
```

This says that a process node in the same partition as a distributor node, may convert it to a process node by making the distributor node adjacent to it on the right.


PI
PO
X
{a}
N
{z}
Y
B
{}
Y
{y}
C
X
{b}
B
{}
Y
B
{a}
N
{z}
C
P1
Pz
links {X Y ...} of node N become secondary links and are deleted
links {B C ...} of node N become secondary links and are deleted
Pa
PO
{a}
N
{z}
M
N
X
{b}
M
{}
Y
B
{}
M
{y}
C
N
M
{a}
N
{z}
distributor node B becomes process node M in new partition Pa
collector node Y becomes process node M in new partition Pz

a. Distributor (B) and Process Nodes c. Process and Collector (Y) Nodes

b. Adjacent Nodes (a) d. Adjacent Nodes (c)

Figure 18. Conservative Merges (Distributor / Collector Adjacency)

```
(6) Pre-conditions    (Figure 18c/d)
    partition(Y)    = partition(N)
    type(Y)         = collector
    type(N)         = process
    inlinks(Y)      = inlinks(N)

    Post-conditions
    adjacent(partition(M) partition(N))
    type(M)         = process (was Y)
    adjacent(M N)
```

This says that a process node in the same partition as a collector node, may convert it to a process node by making the collector node adjacent to it on the left.

Conditions (5) and (6) may require the creation of a partition and may give rise to secondary links. These conditions can also be caused by the merge of non-conservative nodes.

Conditions (5) and (6) may require the creation of a partition if there is no adjacent partition that is between the partition of the distributor or collector node and the partition containing the nodes to which it is adjacent. This usually applies to the first occurrence of this condition. In subsequent occurrences of this condition, the subject distributor or collector node can be placed in the newly created partition when the subject nodes are independent of the node that caused creation of the partition (Figure 18).

Whenever a node is put into an intermediate partition, it has links on both the left and right (indegree > 0, outdegree > 0). This means that all intermediate partition nodes are process nodes. And this means that distributor nodes only appear in the input partition and collector nodes only appear in the output partition.

With conditions (5) and (6) comes the first possible occurrence of secondary links (Figure 18). The procedure for removing secondary links (to maintain the primary link invariant) is applied after every operation and always starts with a single node. Start with the converted distributor or collector node or with the merged node. The procedure (with starting node N) is:

```
-   delete intersect(
            outlinks(N)
            inlinks(N))
    from each list (occurs when two
    adjacent nodes are merged);
-   for the set of nodes, {J K ...},
    referenced in inlinks(N) and the set
    of nodes, {L M ...}, referenced in
    outlinks(N):
    - delete
        intersect(
          outlinks({J K ...})
          inlinks({L M ...}))
      from each list.
    - for the set of nodes,
        {inlinks({J K ...})},
      and the set of nodes,
        {outlinks({L M ...})}:
      - delete
          intersect(
            outlinks(
              {inlinks({J K ...})})
            inlinks(
              {outlinks({L M ...})})
```

```
   from each list.
 - delete
     intersect(
       outlinks({J K ...})
       inlinks(
         {outlinks({L M ...})}))
   from each list.
 - delete
     intersect(
       outlinks(
         {inlinks({J K ...})}))
       inlinks({L M ...})
   from each list.
 - iterate in the same manner until
   partitions PI (no more input
   lists on the left) and PO (no
   more output lists on the right)
   are reached.
```

Non-Conservative Merges. The property of a non-conservative merge is that the matrix interpretation of the sentence after the merge is not identical to the matrix interpretation of the sentence before the merge. At least one element will have been added to the matrix (corresponding to at least one link having been added to the initial bipartite graph).

This is where labels of the same node interact. All output labels of a node are adjacent to all input labels of the same node. This is also where links and labels interact. All output labels of a node, which is adjacent to or reachable from another node, are reachable from the input labels of that other node. Non-conservative merges forge new links between input and output labels by bringing previously unlinked labels into the same node or by adding new links between their nodes.

Non-conservative conditions are formed from the set, (1) - (6), of conservative conditions by weakening either or both of the node type or the node set relation:

> the node type is weakened by changing it from distributor or collector to process,
>
> the node set relation is weakened by changing it from equality to subset,
>
> the node set relation is further weakened by changing it to non-null intersect,
>
> the node set relation is weakened by allowing more than one link -- cases (3) and (4),
>
> combinations of the above.

Nodes which are not considered for merge are:

> Nodes which are in different partitions but which are not adjacent,
>
> Nodes with set relation of null intersect (nodes which have no adjacent nodes in common).

V. Decomposition

Top-down decomposition of a transformation schema process, performed in the software engineering analysis phase, can be viewed as a generate and test activity. The decomposition is a construct, or *model*, that "logically implements" the *process*. The decomposition model is first generated (a synthesis activity) and is then tested (an analysis activity). The analyst validates the model (doing the right thing?) against requirements, user expectations, experience with and knowledge of the problem domain. The analyst verifies the model (doing the thing right?) through checks on completeness and consistency (for example, level balancing) -- this is the area in which most of today's analysis tools are being built.

This algebra addresses the model building synthesis activity and complements the verification analysis techniques, leaving the analyst free to concentrate on the most difficult part of the analysis task, the semantic and pragmatic issues (validating the analysis). The analyst guides the decomposition through a process of matrix assignment, recognition of decomposition functions and decomposition direction.

As presented so far, the algebra applies to either the DFD or the CFD component of the transformation schema. To solve real -time problems, the algebra is applied in a larger framework. In this framework, flows are distinguished as data flows or control flows (with subtypes that will be ignored here since they do not affect the algebra). The framework is:

- input and output flows are each grouped into data and control flows in the I/O matrix -- this breaks the matrix up into four blocks (Figure 19),
- the algebra is not applied directly to this matrix, but to parts of the matrix,
- the input data flow to output data flow block (the DFD decomposition matrix in Figure 19) is decomposed first with the algebra in the usual way,
- a single control transformation is formed and all input and output control flows are assigned to it,
- each element in the data flow to control flow block generates a control flow from the data transformation which inputs that data flow to the control transformation,
- each element in the control flow to data flow block generates a control flow from the control transformation to the data transformation that outputs that data flow,

- predicates defined on input and output data flows generate control flows from the control transformation to the data transformation associated with the data flow,
- the control transformation can now be treated as a context process with a control flow to control flow matrix defined for it,
- external input to output control flow information is transferred from the original matrix input control flow to output control flow block (the CFD decomposition base matrix in Figure 19),
- the rest of the matrix is filled in and the algebra is again applied in the usual way -- this completes the decomposition.

OUTPUTS

		data flow	control flow
INPUTS	data flow	DFD decomposition matrix	data flow to control flow
	control flow	control flow to data flow	CFD decomposition base matrix

data flow	data flow input / data flow output
control flow	signal in / signal out, and data flow based input predicate / output predicate

Figure 19. Real-Time Decomposition Matrix

A. Initiating the Decomposition

Decomposition is initiated when the analyst chooses to expand a single transformation schema process. This becomes the context of the decomposition, whether it is a context diagram or a process from a previous decomposition. The matrix for the decomposition is set up by analyzing the problem and deciding, for each input/output pair of the context, if that particular input is ever used to produce that particular output, either conditionally or unconditionally. The analyst then constructs a bi-partite graph from the matrix and writes the initial sentence of the decomposition as a two partition sentence, taken directly from the bi-partite graph.

As an example, suppose that the context is that of a robot controller [11] as shown in Figure 20 (this is the same problem that was solved in [1] with the data flow only method; here, two of the flows, push buttons (b) and lights (x), are specified as control flows to demonstrate the real-time solution framework). The analyst determines from an analysis of the problem that the I/O matrix for this problem is as shown in Figure 21 (in the data flow solution, the matrix looked like Figure 4; Figure 21 was derived from Figure 4 by sorting it into blocks so that it is in the form of Figure 19). From the DFD decomposition matrix block, the analyst constructs a bi-partite graph, Figure 22a, and writes the initial sentence of the decomposition as two partitions:

{A C D E} {Y Z}

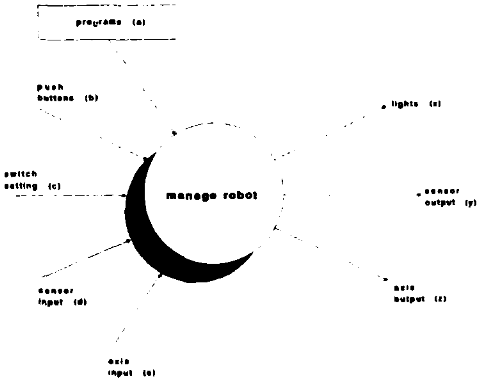


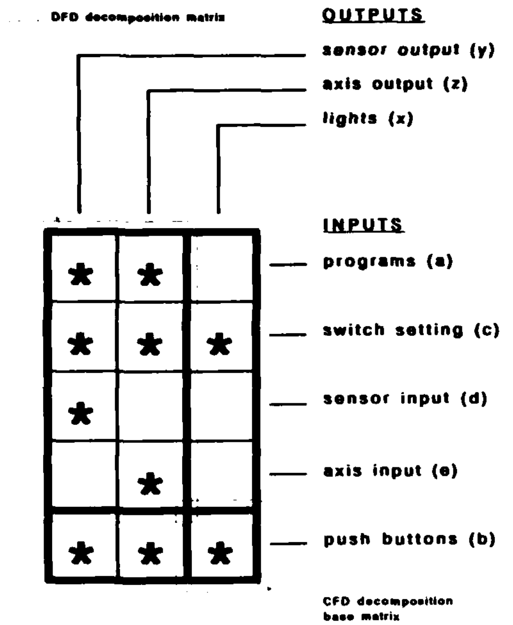
Figure 20. Manage Robot Example

B. Applying the Algebra

The actual decomposition is performed by repeatedly applying the algebra operator to the initial sentence of the decomposition. The operator conditions are considered in the following order:

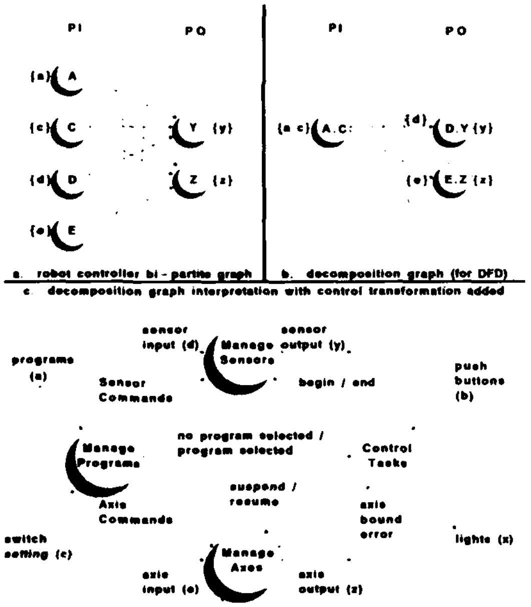
Conservative Conditions
(1) and (2)
(3) and (4)
(5) and (6)

Non-Conservative Conditions
Smallest number of links involved
Smallest number of labels involved



	sensor output (y)	axis output (z)	lights (x)
programs (a)	*	*	
switch setting (c)	*	*	*
sensor input (d)	*		
axis input (e)		*	
push buttons (b)	*	*	*

Figure 21. I/O Matrix for Robot Controller Real – Time Decomposition



a. robot controller bi - partite graph

b. decomposition graph (for DFD)

c. decomposition graph interpretation with control transformation added

The analyst first names the processes, based on input/output flow names
The analyst can then name the local flows, based on the process names

Figure 22 Robot Controller Bi - partite Graph to Decomposition Graph (from the DFD decomposition matrix)

The operato*r* is repeatedly applied to the sentence until the minimal termination criteria is reached (recall that minimal termination criteria means no adjacent distributor-collector node pairs). This is called a *minimal solution.*

The operator is then reapplied to see if the maximal criteria can be satisfied to find a ***maximal solution*** (recall that a maximal solution means all distributor and collector nodes are converted to process nodes). If a maximal solution is found, it becomes the solution; if not, the minimal solution is used. If the minimal solution is conservative, then only a conservative maximal solution is accepted over it. *Conservative* conditions always cause a merge to be performed.

If the minimal solution is a single process, then a *trivial solution* has been found (*occurs if the* bi-partite graph is complete and may occur if it is not connected, although in this case there may be several connected components that can be treated independently). When a trivial solution occurs and flows are not decomposable, the original bi-partite graph may be considered as a solution alternative.

Returning to the robot controller example, the DFD part of the solution goes like this (underscoring highlights the terms for the next operator application):

0. initial sentence (from Figure 22a):

   terms:
   {A C D E} {Y Z}

   (there are three operand pairs here: A and C, D and Y, E and Z. Since *they are independent*, the algebra can merge them concurrently)

   nodes:
   {{} {a} {} {Y Z}
   {} {c} {} {Y Z}
   {} {d} {} {Y}
   {} {e} {} {Z}}

   {{A C D} {} {y} {}
   {A C E} {} {z} {}}

1. conditions: (1) [A&C],
   (3) [D&X, E&Z]
   (Figure 22b)

   terms:
   {A.C} {D.Y E.Z}

   minimal solution (conservative)
   (there is no maximal conservative solution)

   nodes:
   {{} {a c} {} {D.Y E.Z}}

   {{A.C} {d} {y} {}
   {A.C} {e} {z} {}}

C. Interpreting the DFD Result

To produce the initial DFD graph interpretation of the solution, the nodes of the sentence become the graph processes, the node links are paired to produce the graph local flows, and the node labels are the external input and output data flows. Diagram layout is in partition order.

This is where a control transformation is added. The external input and output control flows are allocated to it. The data flow to control flow block, the control flow to data flow block (both from the I/O matrix) and any predicates (defined by the analyst) connect it to the data transformations.

To complete the initial transformation schema, the analyst:

1. replaces the algebra symbols with the actual flow names from the context diagram

2. names each of the transformations, based on recognizing the data transform of inputs to outputs or the control function (from the experience and knowledge of the analyst)

3. names the local flows connecting the transformations (again, from experience and knowledge)

Continuing the robot controller example, if the initial solution sentence from step 1, above, is:

```
     terms:
     {A.C}  {D.Y  E.Z}

     nodes:
     {{} {a c} {} {D.Y  E.Z}}

     {{A.C} {d} {y} {}
      {A.C} {e} {z} {}}
```

then the processes of the interpretation are:

```
     Processes
1.   A.C
2.   D.Y
3.   E.Z
```

and the local flows are between processes:

```
     Local Flows
1.   {A.C}  {D.Y}
2.   {A.C}  {E.Z}
```

The control transformation is now added, with the following:

> the external input and output control flows, bush buttons (b) and lights (c) are allocated to it,
>
> from the data flow to control flow block of the I/O matrix (see Figures 19 and 21), one control flow is generated: from the data transformation that inputs the data flow "switch setting" to the control transformation,
>
> from the control flow to data flow block of the I/O matrix (see Figures 19 and 21), two control flows are generated: from the control transformation to the data transformations that output the data flows "sensor output" and "axis output",
>
> the analyst has defined a predicate called "axis bound error" on the input data flow called "axis input":
>
> ```
>   forsome(i),
>     ¦axis_position(i)¦ > bound(i)
> ```
>
> which generates a control flow from the data transformation that inputs the data flow "axis input" to the control transformation.

The resulting initial transformation schema is shown in Figure 22c. The analyst has named the processes, based on the external input and output flows, and has then named the local flows, based on the process names.

D. Continuing the Decomposition

The control flow transformation can now be decomposed by constructing the matrix that is indicated by the control flows in the initial decomposition, filling it in, constructing the bi-partite graph, writing the initial sentence, and applying the algebra to this sentence. This results in a CFD, just as the data flow context decomposition resulted in a DFD.

Continuing the robot controller example, the 3-by-3 matrix for the control transformation decomposition is constructed using the control flows in Figure 22c and the CFD decomposition base matrix information from the initial I/O matrix (Figure 21). The analyst supplies the rest of the information to fill in the matrix, which is shown in Figure 23.

From this matrix, a bi-partite graph is constructed (Figure 24a). The initial sentence corresponding to this graph starts the control transformation decomposition (as before, underscoring highlights the terms for the next operator application):

0. initial sentence (from Figure 24a):

   terms:
   {$\underline{B}$ $\underline{F}$ G} {$\underline{X}$ V $\underline{W}$}

   (there are two operand pairs here: $\underline{B}$ and $\underline{F}$, $\underline{X}$ and $\underline{W}$. Since they are independent, the algebra can merge them concurrently)

1. conditions: (1) [B&F]
   (2) [X&W]
   (Figure 24b)

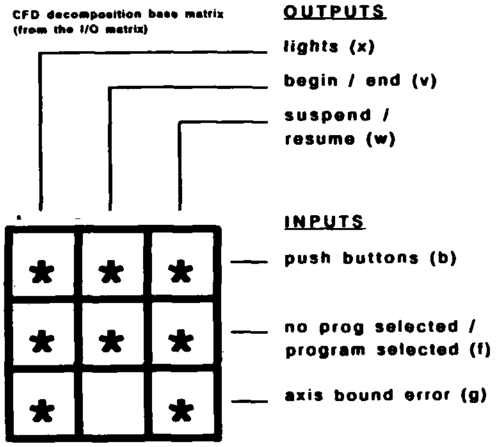
   terms:
   {$\underline{B.F}$ $\underline{G}$} {$\underline{X.W}$ $\underline{V}$}

   (there are two operand pairs here: $\underline{G}$ and $\underline{X.W}$, $\underline{B.F}$ and $\underline{V}$. Since they are independent, the algebra can merge them concurrently)

2. conditions: (3) [G&X.W]
(4) [B.F&V]
(Figure 24c)

terms:
{B.F.V} {G.X.W}

maximal solution (constructive)
(there is no minimal solution)



	lights (x)	begin / end (v)	suspend / resume (w)
push buttons (b)	*	*	*
no prog selected / program selected (f)	*	*	*
axis bound error (g)	*		*

Figure 23 CFD Decomposition Matrix for Robot Controller

E. Interpreting the CFD Result

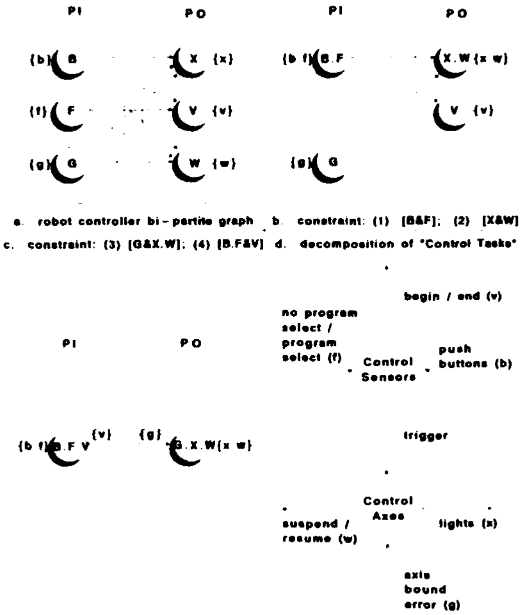
The terminal sentence produces two process nodes:

Processes
1. {B.F.V}
2. {G.X.W}

and the one link connecting them generates one local *flow*:

Local Flow
1. {B.F.V} {G.X.W}

Assigning partitions from top to bottom, instead of from left to right, produces the CFD decomposition of the control transformation (Figure 24d). The analyst has named the processes, based on the external (to the control context) input and output control flows, and has named the local flow based on the process names.



a. robot controller bi-partite graph b. constraint: (1) [B&F]; (2) [X&W]
c. constraint: (3) [G&X.W]; (4) [B.F&V] d. decomposition of "Control Tasks"

Figure 24. Robot Controller Control Transform Decomposition (from the CFD decomposition matrix)

F. Terminating the Decomposition Process

In the overall decomposition process, one of the difficult questions is the determination of when to terminate the process. Using this algebra, the decomposition criteria that specifies the next step in the decomposition process is based on the process to be decomposed and the composition of the flows in to, and out of, the process.

A *primitive process* is one with a trivial decomposition and a *primitive flow* is one with no components (neither a compound data flow, nor a control flow bundle):

Next Decomposition Step	Criteria
process decomposition	non-primitive process
flow decomposition	primitive process and non-primitive flows
terminate decomposition	primitive process and primitive flows

G. Decomposition Quality

This algebra provides a quality metric for decomposition. Based on the decomposition and the matrix interpretations of the initial and terminal sentences, the decomposition may be *trivial*, *optimal* or *feasible*:

Quality	Definition
trivial	decomposes to a single process
optimal	initial and terminal matrices are identical and not trivial
feasible	initial and terminal matrices are distinct and not trivial

Optimal decompositions occur when the decomposition can be accomplished using conservative operators exclusively. There may be both minimal and maximal solutions, or just one solution. The manage robot decomposition example is optimal.

Feasible decompositions have multiple solutions and result when any non-conservative conditions apply (Fig. 15).

H. Handling Decomposition Feedback

All input flows are treated as distinct from all output flows. This means that when a process has an output flow to a store and an input flow from the same store, the flows to and from the store are treated as distinct by the algebra. Non-store input and output flows of the same name are treated as representing different flow iterations, or as before-and-after flow updates, or as being otherwise distinct.

After the decomposition has been completed, a local flow is added to the resulting diagram. It connects the process generating the flow as output, to the process accepting the flow as input (if these are the same process, no local flow is generated). The name of this local flow is the same as the common input-output name.

I. State-Transition Decomposition Overview

The Ward-Mellor methodology [5] calls for final decomposition of all control transformations as state-transition diagrams. When used with the algebra, that means all lowest level control transformations. If the single control transformation added after the DFD decomposition step has a small number of control flows, the CFD decomposition step can be skipped. Conversely, if the control transformation is very complex, it may require more than one level of decomposition to make the individual control transformations manageable (especially if control flows are not bundled at the higher levels). In either case (or in between), the lowest level control transformations are decomposed to state-transition diagrams.

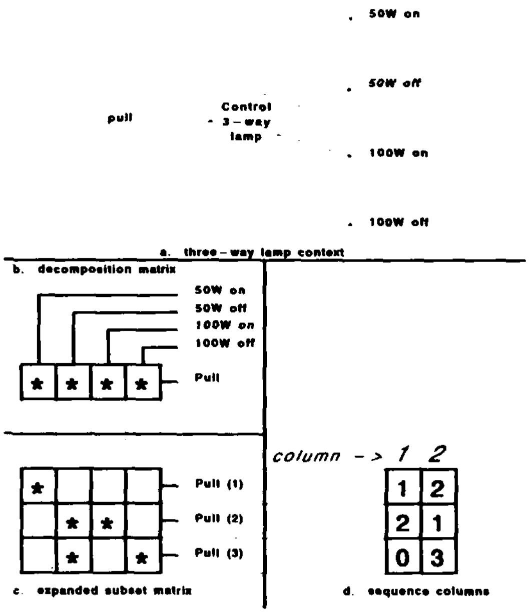
The starting point for this adjunct to the algebra is the control transformation's CFD specification matrix. Additional information will be required from the analyst to produce the state-transition diagram, but there are several points in common with the algebra:

the additional information is about the inputs and outputs only (as it is with the algebra),

the number and arrangement of states and transitions is produced by the procedure, not directly specified by the analyst -- it will change to meet the specification as the specification changes (this is similar to the way that the algebra constructs and changes the processes of the transformation schema to follow the specification),

A very small example, from Ward and Mellor [5], is used to illustrate the process. The example is the state-transition diagram for a three-way lamp:

> A three-way lamp bulb contains two filaments, one of lower power, one of higher power. Assume a lamp bulb with 50 and 100 watt filaments. Successive pulls on the lamp switch chain cycle the bulb from 0W (off) to 50W to 100W to 150W and back to 0W (off) as the individual filaments go on and off (Figure 25a).



a. three-way lamp context

b. decomposition matrix

c. expanded subset matrix

d. sequence columns

Figure 25. Augmenting the Specification for State-Transition Diagrams

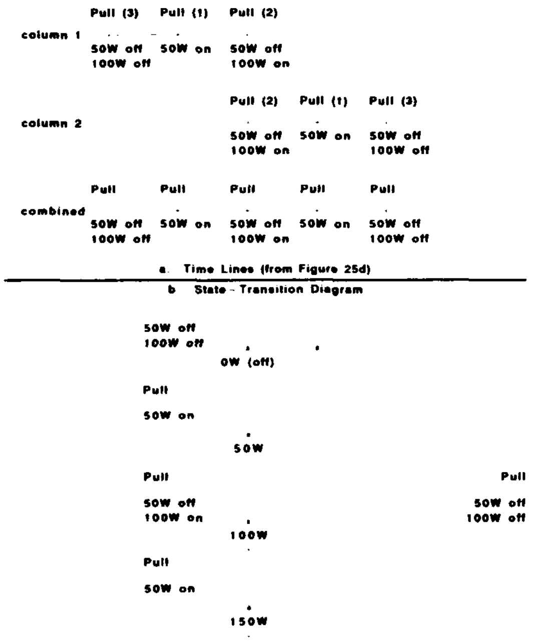
Figure 25b shows the specification matrix corresponding to the context of Figure 25a. The next two steps comprise the additional information supplied by the analyst:

each row of the matrix is expanded so that there is one control flow input row for each allowed subset of control flow outputs in the original line which can occur together,

columns are added to specify all of the sequences in which the input control flows may occur -- each column generates a time line (these time lines are specified independently so that additional specification may be added later without disturbing what is already there).

The time lines are then combined to form a state-transition diagram. The analyst will try to name the states and may iterate to get a (better) solution.

In the three-way lamp example, the expanded matrix rows are shown in Figure 25c and the sequence columns in Figure 25d. The time lines corresponding to the sequence columns are shown in Figure 26a and the state-transition diagram constructed from these time lines in Figure 26b.



a. Time Lines (from Figure 25d)

b State - Transition Diagram

Figure 26. Time Lines and State - Transition Diagrams

It is expected that the complexity of the independent time lines does not go up nearly as fast as the complexity of the states (no combinatorial explosion for time lines, which can be thought of as forming a basis set for the state-transition diagram under construction). One concern about combining time lines is the question of whether they will combine directly for large problems or whether they will need some form of interval logic [12, 13] to resolve ambiguities. Work is proceeding towards a working model in this area.

VI. Tools

The algebra is fairly straightforward for small problems, but becomes difficult to apply as the problem becomes larger. This necessitates implementing the algebra as an automated tool if it is to be used for real problems.

A. A Prototype

The original algebra was implemented as a small Prolog model on a standard microprocessor. Both the manage robot example and a set of test cases were developed and validated by hand. This was used to verify the Prolog model. While this worked reasonably well for the author, the level of tolerance that was required to use a graphics tool that only put out text kept the prototype from being widely tested.

The prototype for this algebra (and subsequent transformations, such as state -transition diagram generation) is being implemented in the C language, in a standard windowing and graphics environment, on a standard microcomputer. This is taking longer to implement than the Prolog model took, but should be received by a wider audience for testing purposes.

B. Industrial Strength Production Tools

Making the algebra usable in a production environment requires not only a graphics interface but a tool environment (the prototype is being designed as a scratch pad to hold down the work and to keep it from unintentionally escaping to de facto product status). There are a number of tools currently available which automate the drawing and maintenance of transformation schemas with which the algebra could be combined. The decomposition steps might appear to the user of such an enhanced package as:

1. the analyst selects a process to be decomposed (using the mouse),

2. the package generates an initial matrix, decomposing data flows (from a dictionary), if necessary,

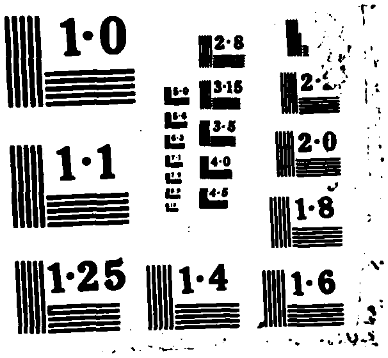
3. the analyst fills in the matrix (using the mouse),

4. the package applies the algebra, generates the graph interpretation, and displays the initial decomposition DFD and single control transformation, with context flow and store names already supplied,

5. the analyst adds process names and local flow names to complete the initial decomposition,

AD-A190 936   PROCEEDINGS OF THE NATIONAL CONFERENCE ON ADA (TRADE   6/7
MARK) TECHNOLOGY (6TH) HELD ON 14-18 MARCH 1988 IN
ARLINGTON VA(U) NORFOLK STATE UNIV VA MAR 88

UNCLASSIFIED   F/G 12/5   NL


1·0
2·8
3·15
3·5
4·0
4·5
2·2
2·0
1·8
1·1
1·25
1·4
1·6

6. the analyst decides to decompose the control transformation as a CFD or as a state-transition diagram,

7. if as a CFD, the package sets up the CFD decomposition matrix, fills in the part corresponding to the CFD decomposition base matrix block in the original I/O matrix, and the analyst fills in the rest of the matrix, which the package decomposes and displays, in the same way it did for the DFD, so the analyst can again name the processes and flows,

8. for state-transition decomposition, the package will present each row of the control matrix to the analyst who will turn off outputs not in that subset,

9. pointing to rows of the expanded matrix with the mouse will build the sequence columns and display the time line under construction,

10. upon completion, the state-transition diagram will be generated and displayed, after which the analyst will try to name the states and may iterate to get a (better) solution.

The user only sees the graphics interface and does not deal with the algebra directly at all.

VII. Outlook

In the near term, work is continuing on the prototype and on the completion and integration of the state-transition decomposition model into the algebra.

In the longer term, transformations will be developed for other parts of the Ward-Mellor methodology. This implies completing the analysis level and defining transformations to and within the design level for generation of task architecture, structure charts within tasks, multi-process and multi-processor models, token modeling, and data model integration.

Other long term objectives are:

the transformation of detailed design information to source code,

reuse of analysis and design information through transformation -- for example, generating a characteristic decomposition for an existing transformation schema by which it could be located later, given an I/O matrix,

the implementation of the inverse of each transformation to implement *reverse engineering*, which is needed not only for source code, but during the development process, if analysis and design information is to remain viable over the life of a product.

The author believes that transformation utilities exemplify the general direction software engineering methods and tools must take if we are to substantially improve the process in the next few years. Methods and tools can be defined and implemented through graphics interfaces to provide an automated sequence of transformations from specification to source code.

Appendix A
Motivation for a New Algebra

In looking at alternatives to the I/O matrix as an interface specification mechanism for the analyst to use in an automated tool, one of the mechanisms considered was to display the input flows on the left, the output flows on the right, and let the analyst connect them. This was rejected over the matrix as a specification mechanism because it required two elements (the input and output flows) to specify what could be said in the matrix with one (toggle matrix element).

In studying this bi-partite graph form, it became apparent that it had advantages over the matrix when used as the underlying algebra. Specifically, it:

reduced the number of operators to one,

provided the best solution without modes (row versus column, and broad versus deep decomposition proved to be artifacts of the representation),

allowed for multiple solutions in a more natural way than equal length rows or columns,

provided a theoretical basis for what were previously assumptions (e.g. validity of the I/O matrix, decomposition optimality),

allowed easy expansion to real-time, and

provided the answer in a format suitable for automatic layout.

Additionally, as the paper progressed, it became clear that:

there were many fewer definitions needed,

some definitions, such as minimal sentence and primary flow, along with new elements not previously considered, such as node and partition, became the basis for invariants,

the conditions for conservation of the specification became specific, instead of probable, as with the previous weak operators, the need for a relative matrix to generate a graph interpretation from a sentence disappeared, and the distinction between primary and secondary links became specific (previously, a decomposition and a matrix were required to generate all of the links, which were then tested for primary or secondary type).

The matrix is retained as the most economical specification representation form.

Appendix B
Equivalence of the I/O Matrix and the Adjacency Matrix

To see that the I/O matrix is equivalent to the adjacency matrix for the decomposition algebra, first consider the general form of a context graph (Figure 2) and construct its adjacency matrix (Figure 27a). If V_i is an input vertex, V_c the context vertex and V_o an output vertex, the only adjacent elements in the matrix are V_iV_c and V_cV_o. No elements in the same partition are adjacent so $V_{i1}V_{i2} = V_{o1}V_{o2} = 0$ for all *i1*, *i2* and all *o1*, *o2*.

	a	b	c	d	e	C0	x	y	z
a	0	0	0	0	0	1	0	0	0
b	0	0	0	0	0	1	0	0	0
c	0	0	0	0	0	1	0	0	0
d	0	0	0	0	0	1	0	0	0
e	0	0	0	0	0	1	0	0	0
C0	0	0	0	0	0	0	1	1	1
x	0	0	0	0	0	0	0	0	0
y	0	0	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0	0	0

a. adjacency matrix of Figure 2

b. reachability matrix corresponding to adjacency matrix of Figure 2

	a	b	c	d	e	C0	x	y	z
a	0	0	0	0	0	1	1	1	1
b	0	0	0	0	0	1	1	1	1
c	0	0	0	0	0	1	1	1	1
d	0	0	0	0	0	1	1	1	1
e	0	0	0	0	0	1	1	1	1
C0	0	0	0	0	0	0	1	1	1
x	0	0	0	0	0	0	0	0	0
y	0	0	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0	0	0

the only part of the adjacency matrix that is not static for this algebra is exactly the part that corresponds to the I/O matrix -- the I/O matrix is equivalent to the adjacency matrix, for this algebra

Figure 27. Equivalence of I/O Matrix and Adjacency Matrix

Now compute the reachability matrix, or transitive closure, (Figure 27b) of the adjacency matrix. The only elements that have changed, versus the adjacency matrix, are that $V_iV_o = 1$. But this is just the complete bi-partite graph of Figure 3, which is equivalent to the I/O matrix. Additionally, $V_oV_i = 0$ remains true due to the DAG invariant. When combined with $V_iV_o = 1$, this is equivalent to the orientations of nodes in partitions being uniform, and this is maintained by the mu-DAG invariant on partitions.

Since no elements in the same partition are adjacent, and the graph is acyclic, $V_{i1}V_{i2} = V_{o1}V_{o2} = V_oV_i = 0$, and the only part of the adjacency matrix that is not static for this algebra is exactly that part that corresponds to the I/O matrix.

The I/O matrix is equivalent to the adjacency matrix, for this algebra.

References

[1] M. Adler, "An Algebra for Data Flow Diagram Process Decomposition," *IEEE Trans. on Software Eng.*, Feb. 1988.

[2] T. De Marco, *Structured Analysis and System Specification*, Yourdon Press, New York, NY, 1979.

[3] P. T. Ward, "The Transformation Schema: An Extension to the Data Flow Diagram to Represent Control and Timing," *IEEE Trans. on Software Eng.*, 12-2, Feb. 1986.

[4] M. A. Gray and M. Adler, "A Formalization of Meyers Cause-Effect Graphs for Unit Testing," *ACM Software Eng. Notes*, 8-5, Oct. 1983.

[5] P. T. Ward and S. J. Mellor, *Structured Development for Real-Time Systems*, vol. 2. New York: Yourdon Press, 1985.

[6] Z. Manna and R. Waldinger, *The Logical Basis for Computer Programming*, vol. 1: Deductive Reasoning. Reading, MA: Addison-Wesley, 1985.

[7] M. N. S. Swamy and K. Thulasiraman, *Graphs, Networks and Algorithms*. New York: John Wiley & Sons, 1981.

[8] C. A. R. Hoare, "An Axiomatic Basis For Computer Programming," *Comm. of the ACM*, vol. 12-10, pp. 576-583, Oct. 1969.

[9] T. Gonzalez and J. Ja'Ja', "Evaluation of Arithmetic Expressions with Algebraic Identities," *SIAM Journal of Computing*, vol. 11-4, pp. 633-662, Nov. 1982.

[10] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Reading, MA:, Addison-Wesley, 1986.

[11] H. Gomaa, "A Software Design Method for Real-Time Systems," *Comm. of the ACM*, vol. 27-9, pp. 938-949, Sep. 1984.

[12] P. C. Fishburn, *Interval Orders and Interval Graphs: A Study of Partially Ordered Sets*, New York: John Wiley & Sons, 1985.

[13] P. Ladkin, "Time Representation: A Taxonomy of Interval Relations," *Proceedings, AAAI-86*, pp. 360-366, Los Altos, CA: Morgan Kaufmann, 1986.

Author Information

Mr. Adler is currently a Senior Consultant and Software Process Designer at Control Data Corporation, Corporate Research and Engineering, in Bloomington, Minnesota.

He has worked on four operating system development efforts between 1965 and 1978, including defining the initial architecture of the SCOPE 2.0 operating system for the CDC 7600.

Since 1978, he has been working in the area of Software Engineering, with current efforts directed toward integrating multiple techniques through the use of novel transformations.

Mr. Adler is a member of the IEEE Computer Society and the Association for Computing Machinery.

Mike Adler, Senior Consultant
Control Data Corporation
Corporate Research and Engineering
2800 E. Old Shakopee Road, HQM234
Bloomington, MN 55440

Development of Innovative Systems in Ada

Thomas J. Wheeler

Software Technology Center
U. S. Army CECOM
Ft. Monmouth N.J.

Abstract: This paper describes an approach to the development of systems in Ada which provide *innovative functionality*, that is those systems whose introduction into use in an organization changes how that organization functions. Since systems of this type change their using organization's behavior, the functionality of the system cannot be adequately specified prior to acquiring experience with the system's use. Current approaches to developing this type of system include: believing that one can write an adequate specification, which doesn't work; prototyping the system, which leads to fielding the prototype; and evolutionary development, which is slow and inhibits real innovation. This paper develops an alternate approach in which a System Specification is developed through a sequence of phases of creation, *formal* specification and analysis; *validating* the concepts and facilities of each specification by methodical construction and formal verification of a prototype of it in an executable very high level language and analysis of simulated use experience.

Introduction

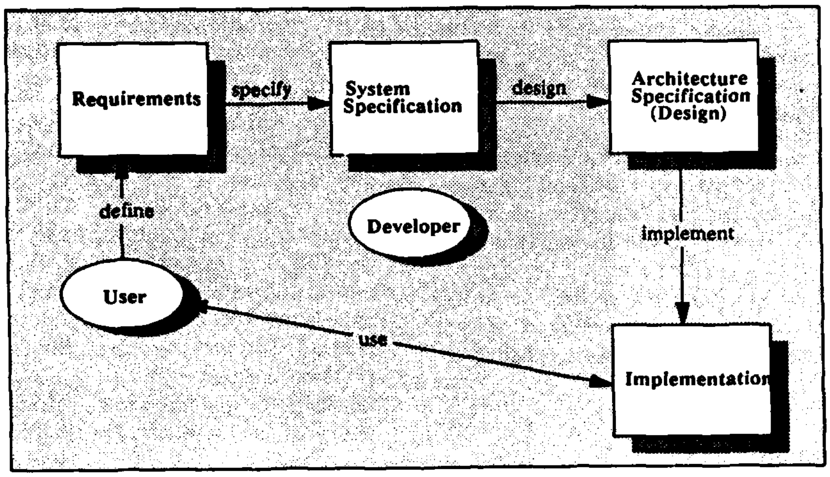
The current life cycle model for the development of software systems[1] provides a linear framework for managing their development starting from their Requirements Document through their Implementation and Fielding[2]. All of the Documentation, which is the "work products" by which the system's development is managed, is in informal, sometimes structured[3], English prose; thus the *semantics* of the system's description is only available to readers of the documentation and is not available for *formal checking* until the implementation is running, thereby providing for formal *observation* and *analysis* of the system's semantics operationally. Unfortunately, this validation of the system's behavior is only available at the end of the development cycle. This is allright if the desired behavior of the system is well known at the start of the development but disasterous for those systems for which it is not. This paper is about the Ada based development of those systems in the second category.

In order to organize the life cycle structure for software systems, it is convenient to categorize software systems into two categories: those for which it is reasonable to expect the system's designers to be able to write an *adequate* system specification based on their intuition and experience with similar systems and those for which that expectation is not reasonable. The systems in the second category are those systems which provide *innovative functionality*; those whose use changes the way the using organization operates, or at least, whose use changes how some individuals do their work; those whose functionality is new and unknown. Systems whose interfaces cannot be specified without observation, by the designers, of the use of the systems by their "real users" also fit into this category. This paper concentrates on the development of systems in this second category by investigating a development strategy, or life cycle model, which addresses the problem of *developing* adequate *system specifications* by providing a formal specification method and a formally assisted *validation* capability for the system specification process.

All of the current approaches to developing this type of system are variations on the cannonical "define requirements, specify, design, implement" life cycle diagrammed in figure 1.

Current Approaches In the first of today's approaches, the "egotistical" approach, the developers assume that they have adequate experience and intuition to design the system and the system is developed, more or less, according to the diagram. The validation of the system's functionality (and the verification of the developers' competance) must wait for the tested implementation to be introduced into operation. As the time between system's conception and system's completion and use is usually rather long, the customer must have great faith in the developer. One would not wait for the completion of a building before deciding whether it was acceptable; but we do that for software. As the first version of the system is usually not viewed by the customer as satisfactory, the "maintenance" phase must be (*mis*)used to get these systems into a condition where they can be at all useful.

An alternate approach is to develop a running prototype of the system. Using the developer's analysis of the intended system's requirements and initial intuition, a running prototype is quickly created by an organized method, eg. the UNIX® "shell scripts and pipes" method[4], or an ad-hoc method, eg. the "hacking" method; LISP methods can fall into either category depending on the mental set of the programmer. This prototype is provided to the users for a trial use period, in which the developers acquire feedback as to the correctness and quality of the prototype's functionality. This



Current Life Cycle Model (figure 1)

process of "trying things out" continues in an iterative fashion until the developer and customer decide that the prototype is satisfactory[1].

There are two possibilities as to how to proceed at this point; in the first, the customer falls into the trap of thinking that the prototype is an accceptable software system (product) and puts the prototype into operation. The main problem with this strategy is that the prototype is neither robust nor maintainability; one would not think of fielding a hardware "breadboard", one should not think of fielding the software analog.

The second possibility, after the prototype is found to be satisfactory, is that the developer will use the prototype as a "running specification". The problem with this is that developing the real system must then be done by a process similar to "reverse engineering"; the prototype has no real specification, the equivalent of one *must* be explicitly, covertly or implicitly constructed in the process of designing the (implementation of the) system.

The last approach, termed the "evolutionary" approach, is a simplification of the Family of Systems[5] approach, in which one starts with the "minimum useable system" and adds functionality until the system is satisfactory. Each step is done using the cannonical life cycle, with each new piece either being added on to the current version or thrown away. If the pieces of functionality are small and orthagonal enough, this strategy works reasonably well.

The problems with this approach are both practical and theoretical. The practical problems are twofold: the first is that the increments are neither small, because it would take too long to build up to a reasonable functionality, nor orthagonal, since discovering "building blocks" which can be combined to build functions is a difficult problem in general. The second is that is is difficult to introduce innovative concepts or facilities, since the currently existing concepts and facilities, at any time, provide a conceptual and implementation *inertia* hindering the introduction of different ways of system functioning.

The theoretical problems are also twofold: first, each of the increments is a software development in itself; if the increments are substantial, one has helped, since each is smaller than the origional problem, but each increment needs a development strategy and none of the strategies mentioned so far is adequate. The second is that evolution only produces adequately constructed systems[6]; there is no design process of the system's overall structure, which evolves over time, it should not be surprising that the final system suffers from a lack of structure.

This is not to say that the prototyping strategy and the evolutionary strategy are not without merit. As will be seen, a variation of the prototyping strategy forms the basis for the life cycle that is developed here, but with a shift in emphasis from the development of the prototype to the development of the specification. The evolutionary development strategy is an excellent implementation strategy, in that, after the first increment is completed, which is early, there is always something working; but it should only be used as an implementation strategy, not a requirements development strategy.

This paper develops a strategy for developing a system's specification through the use of *formal specification techniques* coupled with translation of a formal system specification into a verifiably "correct" prototype. The concepts and facilities embodied in the prototype are then validated by controlled use by the "real" users of the prototype. The feedback, both positive and negative, from the prototype's use, guides the next iteration of the

• UNIX is a registered trademark of AT&T Bell Laboratories

[1] The use of the term "satisfactory" rather than "acceptable" connotes that, in use, the system provides a level of functionality which the users view as helping them accomplish their function, as opposed to meeting some predefined criteria.

specification and its corresponding verified prototype. The feedback helps this process converge on a system which *satisfies* the customer. It is important to note that it is the specification and not the prototype which is the *product* of each iteration.

The paper begins, in the next section, by discussing the formal specification method and introducing the example used to illustrate the method. The translation and verification techinque is described and illustrated in the third section followed by a description of the requirements development method using these techniques in the fourth section. In the last section, the relation of this method to the rest of the life cycle and other conclusions are discussed.

Formal System Specification

As the approach taken here is entirely dependent on formal specification, the Specification Language, its method of use and *formal* System Specifications are defined in this section. The specification method and language are meant to fit well with the model of programming which makes use of explicitly defined modules interconnected together as components to form a system, as is used in languages like Ada®[7] or MODULA-2[8]. This model of programming is very intuitive and matches the style of hardware construction where systems are constructed out of subsystems, which are constructed out of components, etc. In this model, components are explicitly and individually constructed out of smaller components and connected to other components to form the system (or next larger subsystem).

The specification language is meant to be natural and intuitive for programmers to use and therefore the syntax and style of the language are similar to those of modern programming languages. The language has, for instance, modules, types, objects, statements and operations, but these are defined in terms of mathematical concepts rather than in terms of execution of a computer; modules are specifications, types and objects are defined in terms of set theory and statements and operations are assertions.

Specifications are *modular*; they consist of a name, a set of parameters, a set of declarations of types, objects and/or other specifications and a set of operations. Each of the types and objects are defined by a name and a signature. The operations are defined by a name, a signature, a precondition and a postcondition. Specifications are related to other specifications by importing other specifications or by nesting other specifications as local declarations. The importing of specifications allows the use of the facilities defined in the other specification by entities within the importing specification, as well as by other specifications which import that specification; the nesting of other specifications within the set of declarations again allows the use of the defined facilities, but only within the enclosing specification.

The specification language, its concepts and its components will be explained in terms of example specifications which should illustrate and motivate the language and its usage. A specification has a name and collects together a set of entities which are intended by the specification's author to be thought of as either a concept used in, or a component of, the system. Since there are a number of ways of expressing a particular concept as well as a number of concepts closely related to any particular concept, the language provides a flexible means of expressing concepts. The simple examples given here will illustrate the general ways of writing specifications in the language, while illustrating the specification language.

The first example (ex1) is a stack module which allows items, integers in this case, to be PUSHed onto or POPed off of the STACK, which is an object internal to the module.

```
INTEGER_STACK_MODULE: SPECIFICATION
   INTEGER_STACK: (I1,I2,..In);                --a tuple
      where I1,I2,..In : Integer and 0≤n;
      Size(INTEGER_STACK) ≥ 0 ;                --invariant
   PUSH:( INTEGER_STACK, I:Integer → INTEGER_STACK)=
      Post  INTEGER_STACK' = INTEGER_STACK & I;
   POP:(  INTEGER_STACK → INTEGER_STACK, I:Integer)
      Pre  INTEGER_STACK = ( I1,I2,..In),
           INTEGER_STACK ≠ ();
      Post  I' = In.
            INTEGER_STACK' = ( I1,I2,..In-1);
   EMPTY:( INTEGER_STACK → Result:Boolean) =
      Result'= ( INTEGER_STACK = () ); --empty tuple
end INTEGER_STACK_MODULE;
                  (ex1)
```

This module, called "INTEGER_STACK_ _MODULE", specifies a stack of integers by declaring an object, ie. a variable containing the state of the module, called "INTEGER_STACK" which is defined as an n-tuple in which each element is a integer. There is an assertion, the invariant, which states that the stack can never have less than zero elements. There are three operations declared which are "applicable" to the stack, the first of these "PUSH" takes the current value of the stack object and a integer parameter and produces a new value of the stack. The term "INTEGER_STACK" in the signature of the PUSH operation refers to the *actual* INTEGER_STACK object defined on the second line of the specification, whereas "I" is a *formal* parameter which refers to the value of whatever *actual* parameter is supplied when the operation is invoked; "global" variables (objects) are explicitly referenced in signatures in this language. The PUSH operation causes the value of the stack to be equal to the previous value of the stack, with the integer (I) concatenated (&) to the right end of the stack. This is stated by the post-condition of the operation, where INTEGER_STACK' denotes the value of the stack after the operation is complete. Note that in the post-condition,

® Ada is a registered trademark of the US Government (Ada Joint Project Office).

INTEGER_STACK and INTEGER_STACK' denote the value of the *actual stack* before and after the operation invocation respectively, while "I" denotes the value of the *actual parameter* supplied in the "call" of the operation.

The second operation illustrates the use of a pre-condition defining the conditions required before the POP operation can be invoked. The pre-condition states that it is expected that the stack before the operation invocation will contain a number of integers designated as I_1 through I_n and that that number will not be zero. This pre-condition is a compound statement and illustrates two of the type of statements which are used in pre-conditions; the first is an example of a condition being used for a local declaration of a value that an object is stated to have so that it may be referred to in the post-condition thereby allowing the effects of the operation to be defined. The other is an example of a condition which is expected to be checked by the pre-condition, with failure not allowing the operation to be invoked. The post-condition is a compound condition in which the effects of the operation on both the stack and the parameter are defined by statements referring to the symbolic value of the stack declared in the first, declarative, part of the pre-condition.

The use of a condition for the purpose of introducing a local declaration makes sense if one thinks of the definition of an operation (a pre-condition, post-condition pair) in terms of the phrase "if the pre-condition (is true) then the operation occurrs causing the post-condition (to become true)"; in this case, "(1)if the stack *is considered* to contain $(I_1,I_2,..I_n)$ before POP is invoked (*the first pre-condition*) and (2) if the stack *is* not empty (*the second pre-condition*); then the operation POP is invoked causing (1) I to contain I_n (*the first post-condition*)and (2) I_n to be removed from the right hand end of the stack (*the second post-condition*) leaving it containing $(I_1,I_2,..I_{n-1})$."

The last operation is a function which provides(returns) a boolean result based on the current value of the stack. Note that since the only statement in the post-condition is a definition of the value returned, the word "post" is not necessary.

The next example(ex2) is less academic and is closer to the type of specifications that would be used in specifications of systems. This specification defines a (printing) terminal which has line input and output. It reads a sequence of lines from the Input (keyboard) and prints a sequence of lines on the Output (printer). The terminal module is paramaterized by text sequences which can be formally defined, but for now can be thought of as providing definitions of text lines and sequences of text lines, a special syntax for sequences ("*<first & <rest>>*" is a sequence of lines with the leftmost line labeled "first" and the rightmost subsequence labeled "<rest>") and operations applicable to sequences (eg. "&": concatenation of lines). The terminal object which this module declares is defined as a pair of sequences of lines labeled "<Output>" and "<Input>" which are meant to represent, or model, the real terminal's printer or screen and keyboard respectively. As this module is meant to provide for input and output of lines, the two operations provide for reading the first (leftmost) line from the (temporal) sequence of the lines which are going to be typed by the user in the future and the printing of the last (rightmost) line of the lines which have been output in the past.

```
TERMINAL_MODULE: (Text_Sequences) SPECIFICATION
 TERMINAL: (<Output>,<Input>);--(<past output>,<future input>)
   GET: (TERMINAL.<Input> → TERMINAL.<Input> ,
                                   Result:Line| EXCEPTIONS)=
     Pre  TERMINAL.Input  = <First:Line & <Rest>> ,
          <First & <Rest>> /= <> if_fails raise( EOF);
     Post TERMINAL.Input' =  <Rest> ,
          Result' = First ;
   PUT:(TERMINAL.<Output>,L:Line→TERMINAL.<Output>)=
     Post TERMINAL.<Output>' = <Output> & L ;
  end TERMINAL_MODULE;
                 (ex2)
```

The "EXCEPTIONS" declaration and the "if_fails" statement allow specification of failures or errors in the system's operation.

The main example of this section showing the makeup of a system specification is the following example system, which will be used as a running example, illustrating the method, in the rest of the paper. The example is a text editor which is a version[2] of the UNIX™ Ed editor small enough to fit within the scope of the paper, yet substantial enough to allow illustration of the concepts of this paper.

The editor specification consists of two main modules, the environment within which the editor is to be specified and the editor module which defines the behavior of the editor. The environment module defines (contains) models of the terminal, in the Terminal module, and the file system in the Files module. The overall structure of the specification is shown in figure 2.

Here the **Editor_A_Spec** contains the **Editor** and **Environment** specifications, the latter of which contains the **Terminal** and **Files** specifications. The Editor specification uses the Environment specification.

In the specification language, the specification is paramaterized by the library module "text_sequences" which defines lines and sequences of lines. The two main modules which make up the specification are nested within the specification. The Terminal and Files specifications are further nested within the Environment specification. The editor specification module is parameterized by and thereby imports the environment specification. An indented list of the specification headings shows this textual structure(ex3).

[2] This example is what is refered to as the *minimum useful version* of the system[9].

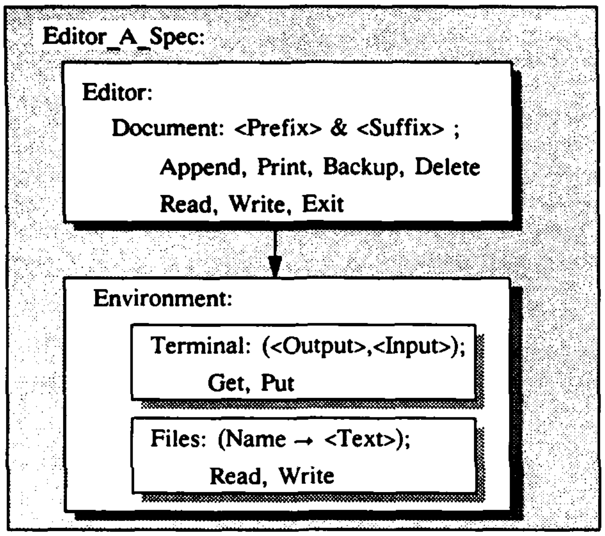


Figure 2

```
EDITOR_A_SPEC: (Text_Sequences) SPECIFICATION
        ENVIRONMENT: SPECIFICATION
            TERMINAL_MODULE: SPECIFICATION
            FILES_MODULE: SPECIFICATION
        EDITOR: ( ENVIRONMENT) SPECIFICATION
                 (ex3)
```

The terminal specification was given above; the Files specification (ex4) defines (models) the behavior of a simple file system by a partial function from file names to sequences of lines contained in those files and READ and WRITE operations on the function.

```
FILES_MODULE: SPECIFICATION
 FILES    : ( Name: STRING ⇸ <Text> ) ;
 Exception_Names contains NonExistent ;
   READ: ( Name: STRING , FILES → <Text>|EXCEPTIONS)=
     Pre  (Name, Any_Text) in FILES if_fails raise(NonExistent);
     Post <Text>' = FILES( Name);
   WRITE:(Name:STRING,FILES,<Text>→FILES|EXCEPTIONS)=
     Pre (Name, Prev_text) in FILES  if_fails raise(NonExistent);
     Post FILES' =(FILES-(Name,<Prev_text>))U(Name,<Text>);
end FILES_MODULE;
                         (ex4)
```

The actual editor specification defines (models) the behavior of the editing system by importing the environment and defining an internal document, to be used as a conceptual model, as a concatenated pair of sequences of lines and defining operations on all of those objects which give the behavior of the editor. The "declarative" part(ex5) of the editor specification module gives the initial conditions of the imported modules, defines and initializes the document within the editor and defines the members of the set of possible exceptions.

```
EDITOR: ( ENVIRONMENT) SPECIFICATION
   INITIALLY:    TERMINAL.<Output> =  <> ;
                 FILES = current file system state;
   DOCUMENT : <Prefix>&<Suffix> ; --> <lines up to and
   including the current line>&<lines following the current line>;
   INITIALLY:    DOCUMENT = <> & <> ;
   Exception_Names contains {Missing_text, End_of_Doc,
                 NonExistent,  At_Line_0, Del_Line_0,  EOF};
                 (ex5)
```

The rest of the Editor specification defines the operations which can take place when using the Editor system. Each of the operations has a pre-condition which defines the circumstances under which the operation is invoked and the circumstances under which an exception occurrs while invoking it and a post-condition which defines the effects of the invocation. An example operation specification which defines the behavior of **appending** a line input from the user's terminal after the current line in the document which raises an exception if an unexpected end of file is encountered is (ex6).

```
APPEND : (TERMINAL , DOCUMENT -> TERMINAL ,
                              DOCUMENT | EXCEPTIONS)=
   Pre  TERMINAL = (<Output>, <C_L:COMMAND_LINE &
                                      Text:Line & <Rest>>),
        C_L.CMD = 'A';
        <Text & <Rest>> /= <>   if_fails raise(Missing_text);
   Post TERMINAL' =(<<Output> & C_L >, <<Text> & <Rest>>);
        TERMINAL'' = (<<Output> & C_L & Text>, <Rest> ),
        DOCUMENT.<Prefix>'' =DOCUMENT.<Prefix>&<Text>;
                    (ex6)
```

The "APPEND" operation requires, in the pre-condition, that the command on the command line be "A" and that there be some text after it on the Input. The post-condition expresses the operation's actions in two steps, thus both the " TERMINAL' " and the " TERMINAL'' " in the post-condition; the first line of the post-condition is not really needed, but this style models the intuitive understanding of the operation, first the operation's name is echoed to the output and then the text is, and makes the proof more straightforward.

Proof Method

Correctness of a specification with respect to another specification is defined in terms of a mapping of the values of the objects of the first specification, usually thought of as an implementation, onto the values of the objects in the second specification and mappings of the transformations between states of the first by operations of the first onto transformations between states of the second by operations in the second[10] in accordance with the diagram in Figure 3.

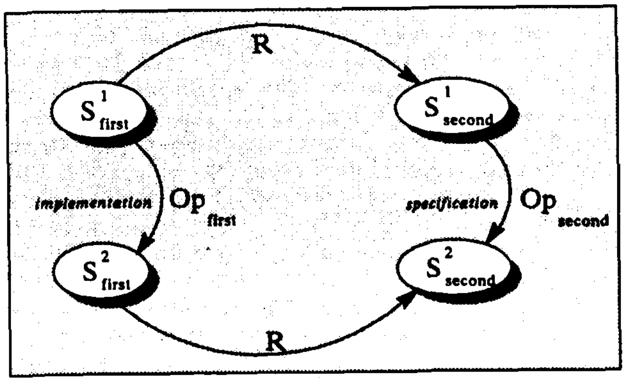


figure 3

The mapping, "R": the representation's map, is an abstraction function. The first specification, usually an implementation, is defined to be correct with respect to the second if the above diagram commutes or equivalently if:

$$R(Op_{first}(S^{1}_{first})) = Op_{second}(R(S^{1}_{first}))$$

for all S and Op in the first and second specifications. The first specification is *correct* with respect to the second, or *correctly implements* the second, if the first is an abstraction of the second; ie. if there is a homomorphic map from the (objects and operations of) first specification to (those of) the second.

Often, in the case when one of the specifications (the second) is a system specification, the operation referred to as Op_{first} above is a sequence of operations, sometimes referred to as a "Path". An example of this will be seen later. The construction of the path equivalent to a specification's operation through the implementation or architecture specification is a ramification of the implicit invocation character of system specifications written in the style described above, which allows us to separate the specifications of the aspects of system behavior and system structure. This correspondence of path to operation is a formalization of the concept of "traceability"; it is a description of, as well as an aid in, allocating system functions to system components.

Formally Assisted Prototyping

The availability of a *formal* System Specification makes possible a way of prototyping the system's functionality which avoids the problems associated with the rapid prototyping methods described earlier, viz. the problem of "fielding" the prototype and the "running specification" problem. The key to the realization of this possibility is the availability of "very high level" languages, through which the specification can be quickly and faithfully implemented. In this approach, we define a translation scheme to guide a translation of the system specification into one of these languages, such that there is a one-to-one correspondence of most of the concepts embodied in the specification in the "very high level" program. We then *verify*, formally, that the program implements the system specification and then *validate* it by having the system's prospective users make use of the prototype system in a set of controlled experimental scenerios to ascertain whether our design is effective.

The process of designing, translating, verifying and validating repeats iteratively improving the design of the system until the users are *satisfied* that the behavior of the prototype is effective, and thus satisfied with the system specification, which was formally verified to exhibit the same behavior as the prototype. It is important to note that is is necessary to verify the equivalence of the prototype and the system specification, since we are really interested in the satisfaction of the specification and the prototype is just a vehicle for that end, *not a product*. This approach to rapid prototyping concentrates on the development of a specification with the prototype being used to assist in that process not on the development of a prototype with a specification being an afterthought (or not thought of). Once any of the prototypes is verified to be functionally equivalent to its corresponding system specification, the validation of the prototype automatically validates that system specification.

The *verification* process for the prototype is relatively straightforward, as will be seen later, using the technique described earlier in the section on proof method. The reason for this relative ease of verification is that the prototyping method is based on a translation scheme which results in the objects and operations in the prototype corresponding extremely closely, usually one-to-one, to the concepts and operations in the system specification; and the verification method is based on constructing a mapping function from the objects and functions of an implementation onto the concepts and operations of the specification. If this mapping is one-to-one, the proof is trivial.

The *validation*, on the other hand, must be carefully thought out and carried out, as the developers and users are attempting to ascertain whether a system, which is new and unfamiliar to the users is helping the users do their jobs better; *but* this is no different than the process of validating the *product* in the standard life cycle; *and* is much earlier in the life cycle and much cheaper; this is the point of this development method.

The method of formal prototyping will be described through the editor example whose specification was outlined and illustrated earlier. The prototype will be programmed in the SETL[11] programming language, a "very high level" programming language based on the incorporation of sets as a built in data type. The SETL language is also a modular language and thus the structure, as well as the contents, of the specification can be straightforwardly portrayed in the prototype program. The Editor prototype(ex7) consists of two modules, one implementing the editor described in the specification and one implementing the exception model used in the specification, along with the built in modules which implement the terminal, which has the same semantics as that in the specification, and the file system, which has the normal record I/O semantics.

```
    Directory EDITOR;                    $ SETL Software Prototype
Program Editor;         imports Exceptions;
                             Procedure  On_Failure_of(Bool_Exp),
                             Macro Raise( Excep, Bool_Exp),
                             Procedure Raised(Excep);
Library EXCEPTIONS; exports Exceptions;
                             Procedure  On_Failure_of(Bool_Exp),
                             Macro Raise( Excep, Bool_Exp),
                             Procedure Raised(Excep);
end Directory; $  (ex7)
```

The Editor prototype implements the specification's COMMAND_LINE, which was a sequence of characters, its COMMAND, which was a character, and its PARAMETER, which was a sequence of characters, with STRINGS; thus there is a straightforward and obvious mapping function for these back onto the specification's corresponding concepts. The DOCUMENT, which was a concatenated (&) pair of sequences of lines in the specification, is implemented in the prototype as a pair of Tupples of strings which again has a straightforward and obvious mapping function for these onto the specification's corresponding concepts. Likewise, the initialization of the DOCUMENT to the empty sequence is implemented by the initialization of the pair of sequences to empty strings. Lastly, EXCEPTIONS in the specification, which is a set of names, is implemented as a set of identifiers, with another obvious mapping to the specification. The declalative part of the prototype is given in (ex8).

```
Program Editor;

    Command_line: STRING;  $ a line. Command_line =
                                    Command + Parameter
    Command : STRING;  $ a character      (+ = concatenation)
    Parameter : STRING;  $ a string

    Prefix : TUPPLE(STRING);   $ DOCUMENT= Prefix + Suffix
    Suffix : TUPPLE(STRING) ;
    Prefix := Suffix := [] ;        $ Initially:  Prefix + Suffix = []

Editor_Exceptions := {Missing_Text, End_of_Doc, At_Line_0,
                                    Del_Line_0, NonExistent};
Exceptions := Exceptions + Editor_Exceptions;
                    (ex8)
```

The operations of the specification are each implemented by procedures in the prototype which have, correspondingly, the same names as the operations. The translation scheme which produces each procedure consists, approximately, of providing one, or sometimes a few, statement(s) for each assertion in the post-condition of the Operation; with Raise statements for each exception raised in the pre-condition. The example operation APPEND is implemented by the Procedure Append, which is shown in (ex9), along with the procedure which implements the exception handler for the exception which can be raised by APPEND.

```
Procedure Append;
    Read(line);
        Raise(Missing_Text,  On_Failure_of( not EOF));
    Print(line);
    Prefix := Prefix + line;
end Procedure Append;

Procedure Missing_Text_Handler;
    Print('[Missing text]');
    Print('E');
    Exit_cmd;
end Procedure Missing_Text_Handler;
                (ex9)
```

In the System Specification, the "invocation" of an operation is implicit in that any operation whose pre-condition is true is eligible to occur, but in the prototype, as it is a program, the invocation must be caused by the program reading and decoding the input to decide which procedure to call. Thus there is an interpreter loop, which again is stylized and straightforward to construct, which controls the invocation of procedures by explicitly decoding those parts of each operation's pre-conditions which determine which operation applies at a given time, in this instance by a **case** on the Command (ex10).

```
loop do
    read(Command_line);
        if EOF then EOF_Exception_Handler end if;
    Print(Command_line);
    Command := Command_line(1);
    Parameter := Command_line(2...#Command_line);
    case Command of
        ('A'): Append;
        ('P'): Print_cmd;
        ('B'): Backup;
        ('D'): Delete;
        ('R'): Read_cmd;
        ('W'): Write_cmd;
        ('E'): Exit_cmd;
        else Error;
    end case;
                (ex10)
```

The detection of raised exceptions is implemented by checking the set of Raised exceptions after the **case** statement(ex11) to see if any are raised and calling the handlers if so.

```
    if Raised /= {} then
        case of
            (Missing_Text in Raised): Missing_Text_Handler;
            (End_of_Doc in Raised): End_of_Doc_Handler;
            (At_Line_0 in Raised): At_Line_0_Handler;
            (Del_Line_0 in Raised): Del_Line_0_Handler;
            (NonExistent in Raised): NonExistent_Handler;
        end case;
    end if;
end loop;
                (ex11)
```

The Path, for the APPEND operation and for the case when there is a premature end of input, which has to be constructed for the proof of correctness, is the concatenation of the interpretation loop body with the statements in the Append procedure, followed by the possible invocation of the exception procedure. These paths are shown on the left side of figure 4.

Proof of the Prototype

As the prototype must be verified to be equivalent to the specification, in order that validation of the prototype also validates the system specification, we must prove that the prototype implements the specification. This proof relies on the simplicity of the abstraction mapping function to make the proof simple. The outline of the proof is shown in figure 4.

Proof Outline of Prototype.Append

```
Abstraction Map R:( PROTOTYPE.EDITOR  -> EDITOR_A_SPEC.EDITOR) =
    Command_line ↦ COMMAND_LINE,
    Command ↦ CMD,
    Parameter ↦ PARAMETER,
    Exception ↦ EXCEPTIONS,
    STD_OUT, STD_IN  ↦ TERMINAL;

Rpath:      Read(Command_line);
            Print(Command_line);
            Command := Command_line(1);
            Append;
                Read(line);
                Print(line);
                Prefix := Prefix + line;                 -R->  APPEND;
or
            Read(Command)line);
            Print(Command_line);
            Command := Command_line(1);
            Append;
                Read(line);
                Raise(Missing_Text, On_Failure_of(not  EOF));
                    Print('[Missing text]');
                    Print('E');
                    Exit_cmd;                            -R->  APPEND;
                        Prefix := Suffix := [];              Raise (Missing_Text);
                        quit;                                Missing_Text_Handler;
---------------------------------------------------------------------------------------
if R(STD_OUT, STDIN) = <<Output>,<"A" & L1 & <Lines> >>
        EDITOR.APPEND'=
            Read(Command_line);
            Print(Command_line);                         -R->  TERMINAL = <<Output &"A">, <L1& Lines>>;
            Command := Command_line(1);                  -R->  C_L.CMD = "A";
            Append;
                Read(line);
                Print(line);                             -R->  TERMINAL = <<Output & "A"& L1>,Lines>;
                Prefix := Prefix + line;                 -R->  DOCUMENT = <Prefix & L1>, <Suffix>;
or
    if  R(STD_OUT, STDIN) = <<Output>,<"A">
        EDITOR.Missing_Text_Handler' =
            Read(Command)line);
            Print(Command_line);                         -R->  TERMINAL = <<Output &"A">, <>;
            Command := Command_line(1);                   -R->  C_L.CMD = "A";
            Read(line);                                  -R->  <Text & <Rest>> /= <>
            Raise(Missing_Text, On_Failure_of(not  EOF));-R->  !Fail => Missing_text!
                Print('[Missing text]');                 -R->  TERMINAL = <<Output & "A" &
                Print('E');                                                  "[Missing text]"& "E">, <>>
                Exit_cmd;
                    Prefix := Suffix := [];              -R->  DOCUMENT = <> & <> ;
                    quit;
```

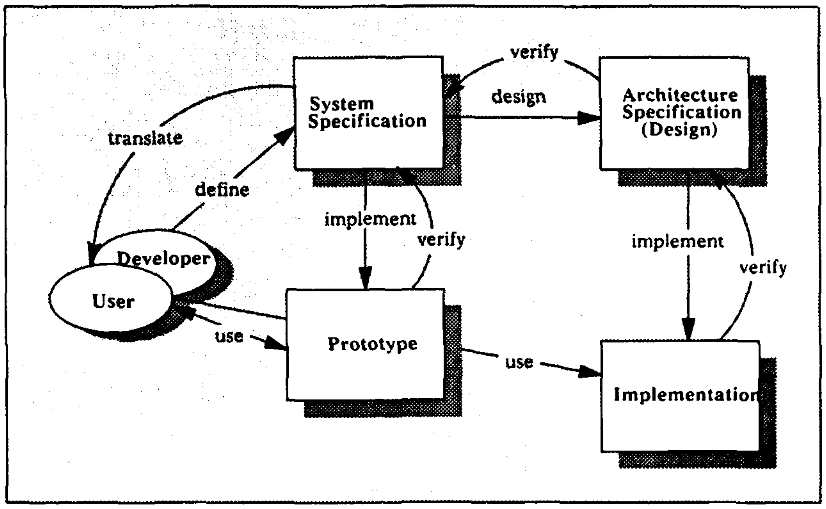
Figure 4

In the figure, the Abstraction Mapping is explicitly defined as a function "R" which maps the prototype editor onto the editor specification and consists of a collection of individual mappings, one for each of the concepts in the specification, which were described earlier, and a collection of mappings of Paths through the prototype onto Operations in the specification. Following the definition of the concept mappings, in the upper half of the figure, the definition of the mapping of the sequence of statements in the prototype onto the operations of the specification is given for the Append procedure; the sequence labeled "R_{path}"[3] maps onto the operation APPEND. This is required, as was alluded to above, since the proof technique needs to form the sequence of object value changes in the implementation so that they can be mapped into the concept values in the specification domain. If the changes to the mapped objects correspond to the specified concept value changes, then the prototype implements the same behavior that the specification specifies[4].

Given the mapping of the sequence of statements in the prototype (its Path) onto the corresponding operation in the specification, the proof is constructed by mapping the resulting values of the objects in the prototype onto the corresponding abstract values of the concepts in the specification domain. The resulting values are compared with the values required by the post-condition of the operation. For instance, the statement sequence "Read(Command_line); Print(Command_line);" reads a line from the terminal's input and then echoes it to the terminal's screen (or printer) and places the value of that line in the (string) variable Command_line; mapping those actions onto the concepts in the specification causes the specification (sequence) variable COMMAND_LINE to

[3] It is the *Inverse* of this R_{path} collection which is the formal Tracability "function" (nb. if there is nondeterminism in the architecture, this is a function onto a *set* of paths, ie. a relation).

[4] More correctly, the post-condition of the mapped path must *imply* the specification's post-condition.



Formal Life Cycle Model

figure 5

contain the line "A<CR>" and the variable TERMINAL to move "A" from the leftmost item on the input sequence to the rightmost item on the output sequence. This is exactly what is specified in the first assertion in the post-condition of the APPEND operation. The rest of the proof outline is similar and equally straightforward. In figure 4 the first half of the proof outline for the APPEND operation is for the case when the operation can succeed and the second half is for the case when the operation fails because there is no more input after the line "A". In all cases, a small number (usually one) of statements in the prototype correspond to, and therefore map their results to, a single assertion in the specification

Life Cycle Model

With the possibility of *validating* the design of a system while writing the system specification, it is possible to have a specification development process become part of the system's development cycle. This leads to a revision of the standard life cycle to provide earlier feedback, a closer working relation between the developer and the users early in the life cycle and, by earlier validation of the system, greater separation of the developer from the user later in the life cycle, where that is desirable. The new life cycle also provides for more control of the product by providing for more verifed work products[12] during the development process.

The new life cycle, shown in figure 5, imposes a prototype implementation between the user and the real implementation, thus shortening the feedback path, and provides for a closer working relation of the developer and the users early in the life cycle, concentration on the development of a specification of a satisfactory innovative system together, rather than expecting the users to know, or rather forsee, what the requirements for one of these are, as is expected in today's life cycle.

The full exploration of the life cycle outlined in figure 5 are beyond the scope of this paper, but it can be seen that this life cycle allows for a greater use of *formalism* and *feedback* than the current one and thus has the capability for *more control* of the development process than today's methods.

Conclusions

The combination of a natural, model based, formal specification method for writing System Specifications and very high level programming languages for which the representation of the concepts used in the specification is straightforward provides the basis for a reorganization of the development process for *innovative* systems which can improve the functionality of that type of system while shortening their development time. The use of methodically created and *formally verified* rapid prototypes in the *development* of system specifications through actual *validation* of those specifications provides the feedback necessary to improve the productivity of the early parts of the development process and thus allow the development of these systems to be cost effective. The *formal verification* of these rapid prototypes and subsequently of the system's architecture and implementation provides the control necessary to make the development of these systems manageable.

References

1. Fredrick P. Brooks Jr., *The Mythical Man-Month*. Reading Mass: Addison-Wesley, 1975.

2. W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," IEEE WESCON, Aug. 1970.

3. D. L. Parnas & P. Clements, "A Rational Design Process: How and Why to Fake It," IEEE Trans. on Software Engineering Vol. SE-12 No 2 Feb. 1986.

4. R. Pike & B. W. Kernighan, "Program Design in the UNIX Environment," AT&T Bell Lab. Tech. J. Vol 63 No. 8, Part 2 (Oct 1984).

5. David L. Parnas, "On the Design and Development of Program Families," IEEE Transactions on Software Engineering Vol. SE-2 No 1 March 1976.

6. Stephen Jay Gould, *The Panda's Thumb*. New York: W. W. Norton, 1980.

7. ANSI/MIL-STD-1815A, *The Ada® Language Reference Manual*, U.S. Dept of Defense, 1983.

8. Niklaus Wirth, *Programming in MODULA-2*. Berlin: Springer-Verlag, 1982.

9. D. L. Parnas, "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering Vol. SE-5 No 2 March 1979.

10. C. A. R. Hoare, "Proof of Correctness of Data Representation," Acta Informatica 1 (1972).

11. R. B. K. Dewar, E. Schonberg, J. T. Schwartz & E. Dubinsky, *Higher Level Programming*. Courant Institute, New York Univ. 1984.

12. S. D. Hester, D. L. Parnas & D. F. Utter, "Using Documentation as a Software Design Medium," The Bell Sys Tech. J. Vol 6 No. 8 (Oct 1981).

Biography

Tom Wheeler is currently responsible for the Research and In-House Technology Base programs in the Advanced Software Technology Office of the Center for Software Engineering, Ft. Monmouth, New Jersey. He has almost completed the Phd. Degree (CS) at Stevens Institute of Technology and has degrees in Physics, Electronic Engineering and Computer Science. His current research interests are: Formal techniques, Programming languages and principles, Data Bases, Distributed Systems and Environments/Operating Systems.

Beyond Ada - Generating Ada Code from Equational Specifications

by
Boleslaw K. Szymanski

Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY 12128

ABSTRACT

Real time mission-oriented embedded systems are much more difficult to design than ordinary software systems. They require highly reliable and efficient implementations to satisfy mission and time constraints imposed by the applications. The Ada language has been design to facilitate real time system software development. However, for many programmers the size and complexity of Ada itself are of concern.

In the assertive programming paradigm, computations are specified as sets of assertions about properties of the solution, and not as a sequence of procedural steps. Solving procedures are automatically generated from the assertive description. Real time programming for mission-oriented systems is supported by equational languages in which assertions are expressed as algebraic equations. Programs written in equational languages are concise, free from implementation details, and easily amenable to verification and parallel processing. The level of programming expertise required to program in an equational language is much lower than the level that is needed by Ada programmers.

The paper describes an implementation of an equational language system which generates highly efficient distributed code in Ada. It also demonstrates how the equational language system can be used in real time software development.

1. INTRODUCTION

Real time system programming is distinct from programming other parallel or distributed applications in that timing constraints are imposed on delays caused by real time programs. The complexity and diversity of skills needed for real time programming have caused extended development times, difficulties in attaining desired reliability and sometimes even a reluctance to undertake maintenance and updating of real time systems. This has motivated development of several programming languages [Brinch 1978, 1981, Martin 1978, Wirth 1977, and most notably Ada, 1978] to make the task easier.

Real time system development can often be simplified if it is done on higher programming level then supported by Ada. Several specification languages [Lamport 1983, Laner 1979, Lee 1986, Milner 1980, Ramamrithan, Teichrew 1977, Zave 1982] have been proposed to this end. Some of these languages support assertive programming paradigm which provide a bridge between the formal requirements of a real time system and the system implementation (for example in Ada). In this paradigm a computation is expressed as a set of assertions about properties of the solution and not as sequence of procedural steps.

In the paper, we discuss the design of an Ada code generator for the assertive language called MODEL [Tseng et al, 1986]. Assertions in MODEL are expressed as recursive equations. MODEL specifications are concise, free from implementation details, and easily amenable to verification and parallel processing.

The MODEL language and system aid or automate the following steps of the software development and maintenance process:

1. Generating high level language code for individual program units. A very high level, nonprocedural language (MODEL) is provided for writing the software specifications. The MODEL compiler uses specifications to generate program code in Ada or other high level programming language (Fortran, C or PL/1).
2. Establishing synchronization and communication between program units executing in parallel. The Configuration Specification Language (CSL) is provided for this purpose. A MODEL subsystem called Configurator generates communication tasks with necessary entries.
3. Testing. An executable model of the system that runs on the host computer is produced by the MODEL compiler and Configurator. This model can be used for testing, debugging and performance study purposes.
4. Documenting. Several reports are generated automatically. The following is a partial list: the system design and structure, individual program listing, generated Ada (or Fortran, C, or PL/1) code listing and timing reports.
5. Static timing performance analysis. Normally, the timing study can be done only after programs in target machine code have been produced and executed. Instead, with the help of a MODEL subsystem called timing evaluator, performance analysis can be done when an individual task has been specified, even on a host other than the target machine.

The paper discusses an implementation of the Ada code generator for the MODEL system. It is organized as follows. In the next section, we describe real time software development using MODEL. Section 3 discusses implementation of the Configurator and generation of communication tasks. Section 4 describes Ada code generation for program units. Finally, the last section offers the conclusion regarding the use of equational languages for real time programming.

2. REAL TIME SYSTEM DEVELOPMENT USING MODEL

In the MODEL approach, the programmer initially partitions the problem into units based on functional affinity. Then, each unit function is described in the MODEL equational language. A series of translators is employed to implement the computation and provide the feedback on performance. This guides the programmer in further partitioning or consolidating parallel units until a satisfactory, locally optimal, performance is reached.

Three software tools were developed to support our approach:

i) A compiler for the configuration specification language in which units' interconnections and a mapping of the parallel tasks onto processors are defined.

ii) A compiler for the MODEL equational language in which individual units are defined. This compiler produces parallel tasks for the respective processors.

iii) A timing evaluator for estimating the delays inherent in the parallel tasks. The estimates are used by the programmer to verify that the time constraints of the developed system are satisfied.

The real time software development process starts after the software system requirement are available. These requirements usually consist of three parts :

1. Functional requirements - defining the functions and subfunctions of the system.
2. Performance requirements - time constraints for time-critical performance of the system.
3. Definition of interfaces with the environment - the layout of the data communicated with the environment.

The programmer begins by dividing the system functions into software units and data files. A function may be carried out by one one or more units, or several related functions may be combined into one unit. The relationship and communications between units are also defined at this point. The program units are in skeletal form, with only the external data structures outlined as files. The programmer can now use the Configurator to verify global system consistency and completeness.

Next, the programmer composes the unit specifications independently for each program unit in the MODEL language. The MODEL compiler processes each unit separately, performing completeness and consistency checks within each unit, and in the absence of errors, generates an Ada program to perform the task of that unit. The user can now employ the Timing Evaluator on each generated Ada program to verify whether the time constraints associated with the corresponding program unit are satisfied. The Timing Evaluator produces a Timing Report for each program unit that provides information on time delays between instances of input and/or output in the unit. The user has to provide certain timing data of the target machine to the Timing Evaluator for it to generate the Timing Report.

The programmer may also have to check if global time constraints are met by adding individual unit delays in a path of the configuration to obtain the overall delays between critical events involving multiple units. If some of these constraints are not satisfied, the programmer may have to modify the configuration of the entire system by partitioning some units to obtain a greater degree of parallelism.

Once all the program units have been satisfactorily processed by the MODEL compiler and the timing evaluator, the programmer uses the Configurator to synthesize all the system components (units and data files) into an integrated system. The user composes the system by specifying a configuration of units and files in the Configuration Specification Language that is input to the Configurator. It then schedules individual program units, synchronizes units that will execute in parallel, generates tasks responsible for exchanging communications, and generates a configuration procedure that will run the Ada programs with maximum concurrency in the host computer's multiprogramming / multiprocessing environment.

Finally, the system can be executed and tested on the host machine. Then, code can be transferred to the target machine for further testing and execution.

3. CONFIGURATION SPECIFICATION LANGUAGE

The Configuration Specification Language, CSL, defines flow of data between program units. Objects of the language are units and files that the units exchange [Shi et al, 1987]. A target/source or consumer/producer relationship between a file (file) and a unit is represented by a directed edge between those objects. When the same file is produced by one unit and consumed by another, then these two units become connected via the file.

Two attributes of configuration nodes are worth of mentioning here. A *unit type* shows whether the unit is:

1) simple - an individually specified unit (default),

2) compound - a group of units for which a configuration is defined separately, or

3) interactive - a human communicating with the system through a terminal.

Files have an *organization* attribute with the following values: sequential (default), indexed, mail and post.

A sequential file is exchanged as one entity. It can be consumed only after it has been entirely produced. Such a file may have only one producer, but any number of consumers.

An indexed file has a variable defined as a key used to define (access) records in the file. There are no restrictions on the order or number of references to such a file made by producers and consumers.

A mail file is a collector of records. It is private to its consumer and therefore it can have only one consumer, but several producers. Records from different producers are accepted by the consumer in order of their arrival.

A post file is a distributor of records to dynamically addressable files. The post file has one producer, and its record include a key used as an address of a destination file. Therefore, it can have any number of edges connecting it to mail files.

An exchange of data between units executed in parallel can be set either through a mail file or a pair of a post and mail files connected together. The goal of our approach is to eliminate timing considerations from real time programming. The user's view of computation is totally static, where computation itself is expressed as a mapping of source data structures onto target data structures. Consequently, our communication primitives are based on (limited) nonblocking 'send' and blocking 'receive'. The producer of the messages continues computation immediately after of messages waits until the message to be read arrives. Such semantics allows the user to treat communications in exactly the same way as other i/o. If the synchronization is needed, it can easily be achieved by adding a 'receive' (in the producer unit) after the 'send' to obtain an answer (or an acknowledgement) from the consumer.

The MODEL compiler, when generating a program for a unit, optimizes the use of the main memory assigned to data, often replacing the entire range of an array by a window, i.e. few elements. When such array has to be communicated to the other units, only that window, i.e. few records at a time, can be sent out. Therefore program optimization causes a producer to store or send as few records at a time as feasible. Similarly, a consumer has also to store and consume a minimum number of records at a time. When producer and consumer processes are concurrent, the post and mail files require a buffer for a limited number of records. This type of data exchange realizes the concept of a pipeline or a stream. The user is not involved in this aspect of program design, however is warned if a file can not be exchanged in that fashion.

The units (processes) and files are the basic building blocks of a system in the MODEL environment. A system can be easily modified by composing a new configuration that includes existing, as well as new or modified, units and files.

The easy modifiability of a configuration supports several development modes. For example, individual units and files may be reused as the system is required to change. Entire independently developed systems may be easily interconnected by adding interfacing processes that convert commonly used variables from the form used in one system to that of the other. Thus, the creation of a new system that encompasses the functions of several old systems would not require designing of a new system.

Ada implementation:

Using Ada as an object language of the MODEL system gave us several advantages over using other high level languages. Ada multitasking and randezvous create a convenient tool for assembling parallel computations. Each MODEL specification is translated into a task. Configuration dependent parts of program units, like interconnections, are encapsulated into separately compiled subprograms and subtasks. A configuration unit, also generated by the configurator, assembles the parallel computation by simply enumerating in its body all the participating units with the 'WITH' clause. Only configurator generated parts of the overall computation have to be recompiled if the configuration changes.

In our design of an Ada implementation of the MODEL specifications, we stressed the independence of computation and configuration descriptions. Units generated by the MODEL compiler need to be compiled in Ada only once. The naming can be local in program units, and the configuration provides the translation of file names in different units. The user is able to select any set of such units and, after providing a configuration specification, generate a configuration unit that will run the entire computation. The configuration unit is compiled separately from MODEL units. Any change in configuration unit does not require MODEL units recompilation. Such solution provides high degree of modularity and supports easy assembling of new systems from existing computational units. Thus, it facilitates fast prototyping and bottom-up development and debugging of real-time systems.

The devised scheme of compilation is as follows:

Each mail file is replaced by a task. This task receives messages from producers, stores them in a queue and then, on the consumer request, moves them to the consumer. Sender and consumer establish randezvous with this task and not directly with each other. Due to the name independence (the same file can be named differently in different program units), sending messages is done through a rerouter procedures which are generated by the Configurator. These procedures contain configuration sensative address tables.

The generated ADA units are as follows:

A. Each MODEL specification of a program unit is compiled by the MODEL compiler into a group of the following packages:

2. Packages for each source mail file in the following format:

```
-- SMAILN - source mail file name
-- UNITN - name of the unit with SMAILN
package UNITN_SMAILN is
task UNITN_SMAILN_mbx is
    entry -- for receiving mail
    entry -- for sending mail
end UNITN_SMAILN_mbx;
end UNITN_SMAILN;
package body UNITN_SMAILN is
task body UNITN_SMAILN_mbx is
-- body of the mailbox (queue of messages)
end UNITN_SMAILN_mbx;
end UNITN_SMAILN;
```

2. A package for a unit procedure with the following structure:

```
-- SMAILN - source mail file name
-- TMAILN - target mail/post file name
-- UNITN - program unit name
with SMAILN_UNITN;  -- repeat
-- for each source mail file in the unit
package UNITN is
procedure UNITN_prog;
end UNITN;
package body UNITN is
procedure UNITN_TMAILN_c is separate;
-- repeat for all post and mail files
procedure UNITN_prog is
   task UNITN_tsk;
   task body UNITN_tsk is
-- code of the MODEL program unit
   end UNITN_tsk;
 begin
   null;
 end UNITN_prog;
end UNITN;
```

B. The configurator produces the following configuration units:

1. For each target post or mail file in the configuration it will generate the re-router in the form:

```
-- TMAILN - target mail/post file
-- UNITN - program unit name
with UNITN; -- repeat
-- for all consumer units
separate(UNITN) -- name of program unit
                -- which contains this file
procedure UNITN_TMAILN_c is
begin
   -- a table of address translation and
   -- case on the value of the table address.
end UNITN_SMAILN_s;
```

2. A configuration unit for invoking the entire computation:

```
-- CONFN is the name of the configuration
with UNITN_SMAILN; -- repeat
    -- for all source mail files
with UNITN_TMAILN  -- repeat
    -- for target mail & post files
with UNITN -- repeat
    -- for all program units
procedure CONFN is
   begin
     UNITN.UNITN_prog; -- repeat
          -- for all program units
   end CONFN;
```

All Ada compilation units are compiled in the following order: A1, A2, and B (order of B1 relative to B2 is irrelevant). Changes in B units affect only the changed package (therefore changing connections between program units and/or adding/deleting program units from configuration is easy and simple). It is worthwhile to note, that during compilation of a program unit no knowledge of configuration in which this unit will participate is needed.

4. MODEL COMPILER

The compilation of an equational specification into an object code consists of four stages: syntax analysis, semantic analysis and checking, scheduling of program events, and generation of the program. The later three stages, relevant to this paper, a summerized below.

Semantic Analysis and Checking:

The compiler translates the specification into a directed graph of data dependences. Use of data dependence graphs to optimize programs, in particular for parallel execution, has been proposed recently in the literature (see for example [Allen et al, 1983], [Ferrante, Ottenstein, and Warren 1984], [Kuck et al, 1981; Waters, 1983]). The distinctive feature of the array graph of the MODEL language is the compact representation of data dependences (a node represents entire array not a single element) and the lack of control dependences (flow of control is generated by the compiler).

Checking the specification and making corrections and additions may be regarded as inferring or propagating attributes from node to node. Thanks to nonprocedural semantics of the MODEL language we were able to implement powerful consistency checks in the compiler. Experience has shown that these checks are effective in locating 80-90% of the errors (not including syntax errors) in development of a program [Szymanski, et al, 1984].

Scheduling Program Events:

In composing a unit specification the user chooses natural and convenient data structures and equations. Typically this choice does not correspond to the most efficient implementation. In addition, the user description of data is independent of the medium of the data and whether it is internal (in main storage), external (secondary storage), or exchanged (communication line carrying messages). It is up to the compiler to map the user's specification into an efficient procedural computer program.

The optimization of the schedule proposed in the MODEL compiler is based on merging scopes of iterations to enable elements of the same or related structures to share memory locations. Usually there are many ways in which components can be merged (for different dimensions), each corresponding to different total orderings of the component graph. The memory requirements of different candidate scopes of iterations serves as the criterion for selecting the optimal merging and corresponding total ordering of the schedule. The selection is equivalent to NP-complete problem of finding a clique with the maximum weight of nodes in an undirected graph. Therefore a heuristic is used [Szymanski, 1987].

Generating Ada Code:

The final step of compilation is program generation that translates the individual entries in the schedule into the object code. In generating Ada code, the MODEL compiler heavily depends on the library of generic procedures for i/o conversions and mathematical operations. These generic procedures are differently instantiated in the generated programs according to the data types used in the specification. The object programs also use overloaded definitions of mathematical functions and operators to keep them independent of the used data types. The generated code use only standard features of Ada. It can be easily added to the existing Ada software. It can also be used as a part of the overall software development process.

5. CONCLUSION

The MODEL equational language provides the programmer with a powerful tool for very-high level, nonprocedural development of the executable system specifications. The MODEL compiler enables rapid prototyping and ensures high level of correctness and consistency checking. Ada, as an object code for the MODEL compiler, provides an efficient implementation tool for parallel execution of the equational specifications. It also ensures smooth synthesis of automatically generated Ada code with the existing Ada software.

Use of an equational language for expressing computations shields the user from considering low level implementation details, like describing input/output operations, loop structure, flow of control in the program etc. Compilation of specifications, including optimization and synchronization algorithms and customized code generators provides the user with efficient implementations of real time systems. Three cooperating components of the MODEL system: MODEL compiler, Configurator, and Timing Evaluator, constitute an integrated software development system that supports rapid prototyping, modularization and comprehensive consistency checking.

Acknowledgement. This work was partially supported by the Office of Naval Research under contract N00014-86-K-0442 and by Computer Command & Control Company. The author wish to express his thanks to Dr. Noah Prywes and Dr. Yuan Shi for discussions of the MODEL to Ada code generator.

References

1. ANSI/MIL-STD-1815 A, Reference Manual for the Ada Programming Language, DOD, Washington D.C., 1983.

2. Brinch, H.P., "Distributed Processes-A Concurrent Programming Concept", CACM, Vol. 21, No. 11, pp. 934-941, Nov. 1978.

3. Brinch, P.H. "EDISON - a Multiprocessor Language", Software-Practice and Experience, Vol. 11, pp. 325-361, 1981.

4. J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The Program Dependence Graph and its Use in Optimization," Proc. 6th International Conference on Programming, LNCS, vol. 167, pp. 125-132, 1984.

5. Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B., and Wolfe, M., "Dependence Graphs and Compiler Optimizations," Proc. 8th ACM symp. Principles Programming Languages, Jan. 1981, pp. 207-218.

6. Lamport, L., "Specifying Concurrent Program Modules", ACM Trans. On Programming Languages and Systems, Vol. 5, No. 2, pp. 190-222, April, 1983.

7. Laner, P.E., Torrigiani, P.R. and Shields, M.W., "COSY - A System Specification Language Based On Paths and Processes," Acta Informatica Vol. 12, pp. 109-158, 1979.

8. I. Lee, N. Prywes and B. Szymanski, "Partitioning of Massive/Real-Time Programs for Parallel Processing," Advances in Computers, vol. 25, Academic Press, New York, 1986, pp. 215-275.

9. Martin, T., "Real Time Programming Language PEARL-Concept and Characteristics", Proc. COMPSAC, 1978.

10. Milner, R., A Calculus Of Communicating Systems, Lecture Notes on Computer Science, Vol. 92, Springer Verlag, 1980.

11. N.S. Prywes, B. Szymanski, and Y. Shi,"Very-High Level Concurrent Programming," IEEE Transactions on Software Engineering, vol. SE-13, no. 8, pp. 1038-1046, September, 1987.

12. Ramamrithan, K. and Keller, R.M., "Specification of Synchronizing Processes," IEEE Trans. On Software Engineering, Vol. SE-9, No. 6, pp. 722-733, Nov. 1983.

13. Szymanski, B., Lock, E., Pnueli, A., and Prywes, N.S., "On the Scope of Static Checking in Definitional Languages," Proceedings of the ACM Annual Conference, San Francisco, CA, 1984, pp. 197-207.

14. Szymanski, B., and Prywes, N., "Efficient Handling of Data Structures in Definitional Languages," Science of Computer Programming, 1987, to appear.

15. Teichreow, D. and Hershey III, E.A., "PSL/PSA: A Computerized Technique For Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. on Software Engineering, Vol.SE-3, No.1, pp. 41-48, Jan. 1977.

16. J. Tseng, B. Szymanski, Y. Shi, and N.S. Prywes, "Real-Time Software Life Cycle with the MODEL System," IEEE Transactions on Software Engineering, vol. SE-12, No. 2, February, 1986, pp. 358-373.

17. R.C. Waters, "Expressional Loops," Proc. 10th ACM Symposium Principles of Programming Languages, ACM, 1983, pp. 1-10.

18 Wirth, N., "Toward a Discipline of Real-Time Programming", CACM, Vol. 20, No. 8, pp. 577-583, August 1977.

19. Zave, P., "An Operational Approach to Requirements Specification for Embedded Systems," IEEE Trans. On Software Engineering, Vol.SE-8, No.3, pp. 250-269, May, 1982.

Dr. Boleslaw Szymanski

Boleslaw Szymanski received Ph.D. in Computer Science from the Polish Academy of Science, Warsaw, Poland in 1976. In 1978 he was a Postdoctoral Fellow at Aberdeen University, Aberdeen, UK. He joined Rensselaer Polytechnic Institute in Troy, NY, as Associate Professor of Computer Science in 1985. Prior to that, he worked at Warsaw Technical University, Institute for Scientific, Technical, and Economic Information in Poland, and University of Pennsylvania, Philadelphia, PA.

Currently, his research interest includes very high level programming languages, parallel algorithms and systems, and large scale, distributed scientific computations. In the past he was also working on database system design and compiler construction and optimization.

Dr. Szymanski is an author of more than 50 scientific publications. He is a senior member of the IEEE Computer Society and member of the Association for Computing Machinery and the Mathematical Association of America.

Changing Module Interfaces in a Software Development Environment

Scott R. Tilley
IBM Canada Ltd. Laboratory

Hausi A. Müller
University of Victoria

Abstract

This paper presents an implementation of the *global interface analysis algorithms*, as proposed by Hood, Kennedy and Müller [HoKM 86]. These algorithms analyze and limit the effects of a change to a basic interface in a software system. The *CMI* (Changing Module Interfaces) implementation is targeted to the Ada and Modula-2 programming languages as an aid for efficient recompilation of module interfaces. *CMI* is an integral part of the *Rigi* software development environment for programming-in-the-large, but can also be used as a stand alone tool for software development and maintenance.

1. Introduction

One of the many problems in a large, evolving software system is the difficulty of changing the interfaces of low level components. A low level component is one that has many clients and/or is near the bottom of the system hierarchy [HoKM 86, Mull 86]. They are usually defined early in the design of a system and, as such, may not benefit from knowledge gained at a later stage of development. The traditional rule in strongly typed, separately compiled programming languages is the recompilation of all modules that use resources provided by the changed definition. This conservative rule is typically implemented using time stamps [Teit 84] and the *make* utility [Feld 79]. In many cases this rule causes the unnecessary recompilation of many modules that do not use the resource that was actually changed in the altered definition. Languages in this category include Ada® [Ichb 79], Modula-2 [Wirt 85], Modula-2+ [Rovn 86], Mesa [MiMS 79], Cedar [Teit 84, SwZH 86], and a number of the recent *modular* Pascal dialects [Josl 87]. A prime example in Ada would be the altering of the **package** STANDARD. A change in this low level interface would affect a great many modules in a large software system, consuming considerable amounts of machine time to bring the system back up to a consistent state.

In a rapidly changing development environment, the time needed to observe the effects of a change to an interface will often cause the programmer to circumvent the compiler's cross-module ch^cking in order to avoid the tedious wait of a [illegible]) long recompilation [Rain 84]. This eva[illegible] [illegible]npiler checks is contrary to the safe modul[illegible] [illegible]osophy that the language provides. Using *CM*[illegible] [illegible] pro-grammer can minimize the time needed for reco[illegible]lation, thus allowing various alternatives in the interface design to be explored before a final decision is reached.

During the maintenance phase cf a software project, the changing of a low-level interface involves the tracking down of all its clients and making sure that they are recompiled in the correct order to maintain a consistent system. In a software development environment such as *Rigi*, aided by *CMI*, the maintainer of a large, complex system can easily decide whether a change in a basic interface can be imple-

Disclaimer: The views and conclusions contained in this paper are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of IBM Canada Ltd.

®Ada is a registered trademark of the U.S. Government, Ada Joint Program Office (AJPO) mented or not by computing the set of affected modules. If the set is acceptable, the change can be implemented, otherwise it can be undone.

The next section of the paper presents some background on pertinent issues of modularization and syntactic interface specifications in the targeted programming languages Ada and Modula-2. Section 3 illustrates the use of *CMI* as a stand alone tool and as part of an integrated software development environment. Section 4 discusses the implementation, and reports on the current status of the project.

2. Modularization and Interfaces

Many programming languages supporting programming-in-the-large, such as Ada, Modula-2, and Cedar, as well as some of the module interconnection languages such as PIC [WoCW 84, WoCW 85a], Intercol [Tich 79]and C/Mesa [MiMS 79], provide facilities for the modular construction and description of large programs. These languages separate the interface specification of a module from its *implementation*. More advanced systems allow for multiple alternatives and/or versions of the implementation to co-exist; the user may select the version desired as late as link time [MiMS 79, LeCM 85].

The benefits of software engineering principles such as modularization, information hiding, and syntactic interface specifications applied to the construction of large software systems are well documented in the literature. However, the recompilation and coordination cost of changing a syntactic interface in large software systems is often prohibitive since too many software components depend on it. This leads to interfaces that are frozen before they are sufficiently explored and tested [Teit 84].

Programmers are unwilling to alter a basic interface since they are not able to estimate the effects on the entire system. Indeed, programmers often try to breach the intermodule type-checking facilities to avoid this (possibly lengthy) recompilation time. This is clearly going against the encapsulation and strong-typing facilities provided by these languages. Typically, contrasting the modification frequency and compilation time of the interface and implementation of a module, the interface is rarely modified (less text) and compiles much faster (no code generation). Nonetheless, consistency checking and recompilation due to the altering of a basic interface is usually much more time consuming than the changing of an implementation part [Tich 82], since numerous other interfaces and implementations in the system may directly or indirectly depend on a basic interface and an implementation part has only local dependencies.

2.1 Modula-2

The modularization mechanisms of Modula-2 include three types of modules: DEFINITION, IMPLEMENTATION, and the main program's MODULE.[1] Every module in Modula-2 (except for the main module) has both a definition part and an implementation part. These correspond to Ada's **package** and **package body**, respectively. The definition module is the interface to its clients.

The most recent version of Modula-2 [Wirt 85] implicitly exports all objects defined in a definition module. The initial definition of Modula-2 required the programmer to explicitly list the exported objects. Both the definition and implementation parts can import objects, but only the definition part can export objects. There is a 1-1 correspondence between definition and implementation modules. A client may import some of the objects provided by a module, or it may import everything in the module. Only identifiers are used in the import and export lists; the Modula-2 syntax does not require one to specify the category of the identifier (e.g., constant, type, variable, procedure).[2] There is an optional QUALIFIED clause that may be applied to exported identifiers. If given, the importing module must always fully qualify references to such objects (in Ada this would be a **with** clause without a corresponding **use** clause).

For example,

```
FROM x IMPORT a,c;      (* specific objects *)
IMPORT y;               (* the whole module *)
```

2.2 Ada

The main mechanism provided by Ada for decomposing systems into modules is the **package** construct. A "module" is a **package** with its corresponding **body**. These packages may be nested; however, the body of the nested package specification must be elaborated in the same body of the corresponding outer package's body or in a **separate** subunit. We do not make a distinction between the two in the algorithms beyond noting the nesting structure. In Ada, one may access objects in a nested package, unlike Modula-2, where only the outermost layer is accessible.

Ada has only two provision levels of resources: all or none. A client gains access to the objects provided by another package through the **with** statement. However, the **with** statement causes all of the package's specifications that are visible to be made available (that is, all those objects that are not **private** or **limited private**). The **use** clause may then be used to provide unqualified access to resources, as long as they do not conflict with already defined objects.

2.3 PIC/Ada

The PIC/Ada language extends Ada by adding **provides** and **requires** clauses to the grammar. The programmer then has greater control over the provision of resources by explicitly providing objects to specific modules. The clients in turn can request the whole package (as in standard Ada) or only the required parts of a package. It also provides an incompleteness construct to alleviate the bottom-up restrictions of Ada development.

3. Using *CMI*

CMI allows for the specification of explicit *provide-require* relationships between modules. The Modula-2 grammar accepted by the parser allows an extension to the EXPORT clause of the form

```
EXPORT [QUALIFIED] TO modules objects
```

where *modules* and *objects* are lists of modules and objects, respectively. This refinement allows stricter control of object provision. If it is not present, then the usual rules of Modula-2 apply. The Ada grammar is extended to allow a more refined importing of objects through a

with package.(objects)

construct. Again, the default Ada rules are used if the standard **with** clause is used.

Both of these extensions can be mapped onto the *CMI* intermediate language. This language is a textual representation of the underlying graph data base stored by *CMI*. Functionally similar to Intercol, it provides a superset of the modular definitions available in standard Modula-2. *CMI* is also similar to the PIC/Ada [WoCW 85b] language in that, though a resource may be made *globally* available, it may also be explicitly provided to only a subset of modules. This facility is available in Anna [LuHe 85] through the provide to clause. This provides an additional level of control over resource provision and requests. An extended Backus-Naur formalism (EBNF) is used to describe the syntax of the *CMI* language depicted in Figure 1 below.
This small grammar allows the *CMI* data base to store the minimal information required for interface analysis. There are no provisions made for version control in the *CMI* system, However, the semantics of the *Rigi* model include version control facilities [Mull 86, MuKl 87].

The user of *CMI* may use the system in two ways: as part of the *Rigi* software development environment, or as a stand-alone tool. As part of *Rigi*, the algorithms are invoked when a basic interface is altered through the *Rigi* editor. As a result, *Rigi* lists the names of the set of affected modules. The list may be used as an indication of the scope of the changes and help predict their effects on the system. When the user has finished editing the interface, the *global interface analysis algorithms* are used to recompile the affected modules.

As a stand-alone tool, *CMI* works as a preprocessor, parsing the source text of the interface and producing an intermediate representation which is recorded in the *CMI* data base.

Suppose we modify an object *xyz* in the package NEAR_BOTTOM, which is near the bottom of the module hierarchy and is part of the *abc* library. In most Ada compilers all units in *abc* would have to be recompiled if they **withed** NEAR_BOTTOM.

[1] The main module is in fact an implementation module [Wirt 87]. Local modules are not considered, since they are not visible outside their enclosing scope which is another local module, a definition module, or an implementation module.

[2] There is no overloading in Modula-2.

```
CmiFile = FileName TimeStamp {ModuleDescription}.
ModuleDescription = ident ":" ModuleType [requires] [provides] END [ident] ".".
requires = REQUIRES {ident ":" ModuleType ["<-" objects]} ";".
objects = ident ":" ObjectType {"," ident ":" ObjectType}.
provides = PROVIDES {[modules "->"] objects [QUALIFIED]} ";".
modules = ident ":" ModuleType {"," ident ":" ModuleType}.
ModuleType = DEFINITION | IMPLEMENTATION | MAIN | LOCAL | UNKNOWN.
ObjectType = PROGRAM | MODULE | TASK | PROCEDURE | CONST | TYPE | VAR | UNKNOWN.
```

Figure 1: Syntax of *CMI* language.

However, if only a few modules in *abc* actually use the object *xyz*, *CMI* would only recompile these few modules.

4. Implementation

CMI is part of the *Rigi* software development environment. As such, it is part of a larger and more complex system for programming-in-the-large. However, its nature of use is such that it can also be employed as a stand-alone tool, though its true strength and potential are only realized as part of *Rigi*.

In part because of time and resource constraints, and in part because of the philosophy of the tool, the existing Ada and Modula-2 compilers were not altered. Instead, *CMI* is implemented as a pre-processor. It parses the source text and extracts the module interconnection information necessary for the global interface analysis algorithms and stores the information in the *CMI* data base. The information extracted is modelled by the grammar outlined above.

The data base is an attributed, directed graph. Each node in the graph represents a module of some type (e.g., syntactic specification or implementation). The links in the graph represent the import and export relationships in the system. In addition, the nodes in the graph are linked through *use* links. This relationship is more specific than the export list; a module *b* is on the use list of *a* iff *b* actually uses and references the objects provided by *a*. This scheme eliminates many redundant recompilations, since many modules often request all of the specification of another, and yet may only use a small subset of the objects provided in the requested module. This is particularly true in Ada, with its *all or nothing* approach to packaging.

In large software systems recursive compilation dependencies occur quite frequently [Clar 85]. *CMI* deals with recursive compilation dependencies by reducing the cyclic graph to a directed acyclic graph (dag). The dag is obtained by computing the strongly connected components of the original graph [Mehl 84]. In most cases, such a strong component consists of a single module. However, for those that have more than one module in them, they are logically bound together and from then on treated as a larger, single module.

When the system is initially built, all files that make up the system are parsed and the modules that they contain are entered into the internal data base. The user supplies the list of files either through command line arguments (for small projects), or by specifying a .PRJ file. This is simply a text file that lists all the files that make up the system, and is updated by *CMI* as new modules are added or deleted to/from the data base. As the modules are being entered, the graph will usually be incomplete (unless the files are entered in the exact order so that dependencies are resolved as they arise). Thus, minimal checking is done while parsing. If an error occurs while parsing the Modula-2, Ada, or *CMI* source, an error is reported at the offending line, and processing returns to the top level. No attempt at error recovery is made. Once the modules have been entered into the system, a complete consistency check takes place. This is to ensure that resources requested by a module from another are also provided, and that there are no conflicting declarations. If the network is consistent, the *use*-links are also put into place at this time.

After the data base has been checked, the system is ready for general use. The user may update the data base by altering and/or deleting existing modules, and by adding new modules. Modifying an existing module does not change the topology of the graph; only the links may be altered. The latter case will involve a local re-mapping of the data base to a graph structure. *CMI* also provides a primitive undo facility for reverting to a previous editing state; this facility comes in handy when the set of affected modules is unacceptably large and the changes have to be undone. This facility is implemented by altering a copy of an existing module, instead of the actual module itself. *CMI* determines the type of change made by comparing the data base entry of the old version with that of the new. This is analogous to comparing the abstract syntax trees of the *provide-require* relationships. The magnitude of the change is characterized as either *inconsequential*, *local*, or *global*, where an inconsequential change has no effect either inside or outside of the module, a local change effects the module but not its clients, and a global change affects the clients [KoKH 86].

The Tichy/Baker algorithm can be used to efficiently deal with local changes [Tich 86]. To compute the set of affected clients for the third case, the global change is to be propagated through the require and provide list of the clients. The immediately affected modules are those that require one or more of the new resources that the changed module now provides (or no longer provides, as the case may be). If the module imports the resource but does not actually use it, a warning only is given; no recompilation is necessary. This situation often occurs in Ada when a **package with**'s another. The whole **package** is made available, but perhaps only a few resources are actually used out of it. However, if the module actually uses one of the changed resources, we must determine if this change will affect any of its exported resources. If this is not the case, then recompilation may halt at this point. Otherwise, we must continue propagating the changes through the *provide-require* relationships of the affected modules. By visiting the set of affected modules in topological order, it is guaranteed that each client is only visited once (i.e., no back tracking).

Unlike Ada, Modula-2 compilers do not usually provide a library manager. Most produce a .sym file from the compilation of a .def (DEFINITION) file [Powe 84]. This mainly consists of time stamp information, and an encoded symbol table. Others simply reparse the .def file every time it is imported by some module [Fost 86]. Neither scheme provides for automatic compilation of the affected modules when a definition module changes. Many users on Unix make use of *makefiles*, but these quickly become complex and hard to maintain for large systems [Wald 84]. *CMI* performs the optimal recompilation checks before invoking the Modula-2 compiler for the necessary compilations. It is here that altering the compiler would improve performance, since the semantic inter-module checks need not be done again by the compiler.

Ada provides a library management system. However, recompilation time can still be improved using *CMI*. Though the Ada library system has much of the same information that *CMI* keeps internally, the recompilation sequence is not as well defined as it could be. Some Ada compilers requires the use of special constructs embedded in the source informing the compiler of the compilation order of the files that affect a package, and that the file is imported by someone else, and will not be compiled unless the main program is. *CMI* can do this automatically, and through the use of the *global interface analysis algorithms* recompile only those packages (files) that absolutely require it. Again, ideally this mechanism should be built directly into the compiler or library manager.

CMI is written in C, and uses the *yacc* and *lex* compiler development tools to aid in the parsing of the source text. The development work was done on an IBM 3090 running VM/CMS, as well as an IBM RT PC running AIX. It has also been tested on an IBM PC/XT running PC-DOS 3.2. We are considering porting *CMI* to systems such as MVS/TSO. As a stand alone tool, the only external interface that needs to change is how files are named on the host system. *CMI* is currently being integrated into the *Rigi* software development environment under construction at the University of Victoria on a network of Sun-3 workstations.

5. Conclusions

This paper presented *CMI*, a practical implementation of the *global interface analysis algorithms*. The current implementation is tailored towards the programming languages Ada and Modula-2; however, the *CMI* implementation could be used as a basis for an implementation of these algorithms for any strongly typed, separately compiled programming language.

Using *CMI*, the time and resources needed to bring a system back up to date after a change to a basic interface is greatly reduced. A user can explore alternate interfaces and estimate the effects of an interface change on the entire system. Therefore, interfaces do not have to be frozen before they are explored and tested. After an interface change, *CMI* determines the minimal set of affected client modules that has to be recompiled and thus avoids many redundant recompilations.

6. References

[Clar 85] Clark, D.D. "The Structuring of Systems using Upcalls," In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, (Orcas Island, WA), pp. 171-180, December 1985.

[Feld 79] Feldman, S. "Make – A Program for Maintaining Computer Programs," *Software – Practice and Experience*, 9(3), pp. 255-265, 1979.

[Fost 86] Foster, D.G. "Separate Compilation in a Modula-2 Compiler," *Software – Practice and Experience*, 16(2), pp. 101-106, February 1986.

[LeCH 85] Leblang, D.B.; R.P. Chase; and G.D. McLean. "The DOMAIN Software Engineering Environment for Large Scale Software Development Efforts," *Proceedings of the IEEE Conference on Workstations*, San Jose CA, November 1985.

[HoKM 86] Hood, R.; K. Kennedy; and H.A. Müller. "Efficient Recompilation of Module Interfaces in a Software Development Environment," *Proceedings of the 2nd ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, (Palo Alto, CA; December 9-11). In *ACM SIGPLAN Notices*, 22(1), pp. 180-189, January 1987.

[Ichb 79] Ichbiah, J., *et al.* "Rationale for the Design of the ADA Programming Language," *ACM SIGPLAN Notices*, 14(6), May 1979.

[Josl 87] Joslin, A.J. "Extended Pascal – Illustrative Features," *ACM SIGPLAN Notices*, 22(5), pp. 18-19, May 1987.

[LuHe 85] Luckham, D.C.; and F.W. von Henke. "An Overview of Anna, a Specification Language for Ada," *IEEE Software*, pp. 9-22, March 1985.

[Mehl 84] Mehlhorn, K. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. Springer-Verlag, 1984.

[MiMS 79] Mitchell, J.G.; W. Mayburg; and R. Sweet. "Mesa Language Manual," Version 5.0, Technical Report CSL-79-3, Xerox PARC, Palo Alto CA, April 1979.

[Mull 86] Müller, H.A. "Rigi – A Model for Software System Construction, Integration, and Evolution based on Module Interface Specifications." Ph.D. Thesis, Rice University, Houston, Texas, COMP TR86-36, August 1986.

[MuKl 87] Müller, H.A.; and K. Klashinsky. "Rigi – A System for Programming-in-the-large," University of Victoria, Victoria, BC, Technical Report DCS-68-1R, September 1987. To appear in the *10th International Conference on Software Engineering (ICSE)*, April 11-15 1988 in Raffles City, Singapore.

[Powe 84] Powell, M.L. "A Portable Optimizing Compiler for Modula-2," *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*. In *ACM SIGPLAN Notices*, 19(6), pp. 310-317, June 1984.

[Rain 84] Rain, M. "Avoiding Trickle-down Recompilation in the Mary2 Implementation," *Software – Practice and Experience*, 14(12), pp. 1149-1157, December 1984.

[Rovn 86] Rovner, P. "Extending Modula-2 to Build Large, Integrated Systems," *IEEE Software*, pp. 46-57, November 1986.

[SwZH 86] Swinehart, D.C.; P.T. Zellweger; R.B. Haggman; and R.J. Beach. "A Structural View of the Cedar Programming Environment," *ACM Transactions on Programming Languages and Systems*, 8(4), pp. 419-490, October 1986.

[Teit 84] Teitelman, W. "A Tour Through Cedar," *IEEE Software*, 1(2), pp. 44-73, April 1984. And in *IEEE Transactions on Software Engineering*, SE-11(3), pp. 285-301, March 1985. And in *Proceedings 7th International Conference on Software Engineering*, (Orlando, FL), (IEEE Catalog No. 84CH2011-5), pp. 181-197, March 1984.

[Tich 79] Tichy, W.F. "Software Development Control Based on Module Interconnection," *Proceedings of the 4th International Conference on Software Engineering*, pp. 29-41, (IEEE Catalog No. 79CH2011-5), October 1979.

[Tich 82] Tichy, W.F. "Adabase: A Data Base for Ada Programming," *Proceedings of the AdaTEC Conference on Ada*, (Arlington, Virginia), (ACM Order No. 825821), pp. 57-65, October 6-8 1982.

[Tich 86] Tichy, W.F. "Smart Recompilation," *ACM Transactions on Programming Languages and Systems*, 8(3), pp. 273-291, July 1986.

[Wald 84] Walden, K. "Automatic Generation of Make Dependencies," *Software – Practice and Experience*, 14(6), pp. 575-585, June 1984.

[Wirt 85] Wirth, N. *Programming in Modula-2*, Third Edition, Springer-Verlag, 1985.

[Wirt 87] Wirth, N. "From Modula to Oberon and the Programming Language Oberon," *Berichte des Institutes für Informatik der ETH Zürich*, No. 82, September 1987.

[WoCW 84] Wolf, A.L.; L.A. Clarke; and J.C. Wileden. "An Ada Environment for Programming-in-the-large," *IEEE Computer Society 1984 Conference on Ada Applications and Environments*, (St. Paul, MN), (IEEE Catalog No. 84CH2083-4), pp. 52-562 October 15-18, 1984.

[WoCW 85a] Wolf, A.L.; L.A. Clarke; and J.C. Wileden. "Ada-Based Support for Programming-in-the-large," *IEEE Software*, pp. 58-71, March 1985.

[WoCW 85b] Wolf, A.L.; L.A. Clarke; and J.C. Wileden. "Interface Control and Incremental Development in the PIC Environment," *Proceedings 8th International Conference on Software Engineering*, (Imperial College, London, UK), (IEEE Catalog No. 85CH2139-4), pp. 75-82, August 28-30 1985.

Scott Tilley is a member of the Languages and Related Products (LRP) group at the IBM Canada Ltd. Laboratory. He is currently completing his M.Sc. degree in Computer Science from the University of Victoria. His research interests include programming-in-the-large, software development environments, programming languages and compilers.

Tilley received his B.Comp.Sc. in Computer Science from Concordia University, Montréal, Québec in 1986. He is a member of the ACM and the IEEE. He may be reached at the IBM Canada Ltd. Laboratory, 895 Don Mills Rd., North York, ON, M3C 1W3.

Hausi Müller received a Diploma Degree in Electrical Engineering from the Swiss Federal Institute of Technology (ETH), Zürich in 1979 and the M.S. and Ph.D. degrees in Computer Science from Rice University, Houston, Texas in 1984 and 1986, respectively. From 1979 to 1982 he worked as a software engineer for Brown Boveri in Baden, Switzerland.

He is currently an Assistant Professor of Computer Science at the University of Victoria, British Columbia. His research interests include software engineering, programming-in-the-large, programming languages, graph algorithms, and computer graphics. He may be reached at the University of Victoria, P.O. Box 1700, Victoria, BC, V8W 2Y2.

ADA ACQUISITION
A TAILORED APPROACH

by George J. Smith
Sandra A. Pryor

Naval Weapons Center
China Lake

This paper addresses what we perceive to be the high risk areas in Ada procurement. We have structured this paper to follow the natural sequence of events including preprocurement activities, Statement of Work writing, and proposal evaluation. Each has a different set of problems and is addressed in a different way. Under preprocurement activities we identify contractual and programmatic "givens" that impact the Ada software procurement. Under Statement of Work writing, we discuss the key aspects of technical management, legal requirements and engineering discipline. Under proposal evaluation, we focus on management evaluation and provide a "tool" to assist in evaluating the alternative Ada software development cycle. In each situation we identify the pitfalls and, where possible, ways to avoid them.

The Naval Weapons Center is the U.S. Navy's principal laboratory for research, development, test and evaluation of airborne weapons systems. Weapons sytems procurements always contain a certain degree of risk. The greatest risk is that the contractor will not meet the requirements established for the weapons system. When the weapons system contains software the risks increase notably. This is partly due to the fact that we cannot specify "good" software by the pound, size, or color. We can describe what it does but we cannot visually picture the software; therefore, requirements, communication, visibility, and control become very difficult. Compound these problems with the requirement to use Ada and our risks of not meeting the requirements usually escalate even higher.

Preprocurement Activities

A high level of activity occurs during the preprocurement phase, from both a contracting aspect and a programmatic aspect. The results of these activities combine to produce an acquisition strategy for procuring or developing the software. Some elements of the acquisition strategy such as Firm Fixed Price contracts and lifecycle support philosophy are "givens" or mandated by Department of Defense directives. These "givens" have a great impact on the software acquisition effort and will be discussed in the following paragraphs.

Programmatic Arena Activities. In the programmatic arena the maintenance strategy or the lifecycle support philosophy of the software must be decided by the Government. If the software will always be maintained by the developer the requirements will be different than the requirements for software that will have organic support by the Government. For example, if the software is to be maintained by the developer we would typically order less documentation and might not require full Ada software library rights. Therefore, it is vital that the maintenance strategy be determined prior to generation of the Statement of Work for the Full Scale Engineering Development Phase.

System engineering is also a programmatic activity which takes place early. Unfortunately, experience has shown us that the system engineer is typically hardware oriented and has neither the time nor inclination to become an expert in Ada and its design techniques. Therefore, system architectures are established according to a functional decomposition rather than an object oriented design. This functional decomposition makes it difficult for the software engineer to map Ada onto the system as it was designed by the system engineer. In the past, software systems were hardware driven (i.e., the software was written to drive the hardware). With Ada, that is no longer the case. The hardware should be designated according to its ability to support Ada. Unfortunately, the hardware oriented system engineer often relies upon hardware specifications which are quoted for the hardware's own assembly language. The specifications will be quite different for running Ada. If the system engineer is not aware of this, he may commit the system to hardware that is neither large enough or fast enough to run Ada. This situation can lead to a very complex system architecture. On the other hand, a system engineer that is only Ada software oriented could design an architecture which no existing hardware can handle. This could be due to size requirements, weight, and environmental constraints. To help remedy this situation, an Ada knowledgeable software engineer should provide input to the system engineer during the design of the system architecture.

Contracting Arena Activities. In the contract arena, basic decisions have to be made such as type of contract and single or multi-phase contract. The Department of Defense presently mandates that we use Firm Fixed Price contracts for all Research and Development efforts. While this is the most favored and cost effective way of procuring standard items when we have prior purchase experience of the same or similar items, it is extremely difficult with Research and Development efforts.

Firm Fixed Price contracts are commonly thought to place the total financial risk on the contractor.

This is only true if we have a solid performance or design specification before entering the contract. That is, we must know precisely what we want to have built. Unfortunately, with Research and Development efforts, especially involving software and Ada, that is not the case.

> The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems... . It is really impossible for a client, even working with a software engineer, to specify completely, precisely, and correctly the exact requirements of a modern software product before trying some versions of the product.[1]

Since we cannot specify exact requirements at the beginning, the contractual specifications are likely to evolve and exceed the scope of the contract. This could produce a need for contract modification and invoke additional cost to the Government. Therefore, the risk to the Government can escalate greatly. In the past we have diminished that risk by using Cost Plus contracts which actually procure the contractor's "best effort" and has built into it a mechanism for handling changes. We know of no immediate solutions for the Firm Fixed Price contract problem. The best we can recommend is to recognize the problem ahead of time and formulate a strategy to deal with the inevitable changes.

Contract Phasing. Another contractual decision involves contract phasing. It must be decided whether to utilize a single or multiple phase contract. Each carries with it certain benefits and certain drawbacks.

There are many factors which determine the contract phasing decision. Phasing of the contract may be greatly impacted by whether the effort is a "new start" or a "Product Improvement Program". If it is a new start, the effort will generally start with a Conceptual Phase, will take longer to get to the fleet and will have less historic data to draw upon. However, a new start provides more flexibility because we are not tied to constraints which are carried over in a Product Improvement Program effort. With a Product Improvement Program we will have a shorter, and sometimes less costly development time frame due to the fact that some of the work done on the original program can be utilized on the Product Improvement Program. The early phases can sometimes be abbreviated. Also, the original program provides a great deal of historic data that can be used on the Product Improvement Program. Unfortunately, a Product Improvement Program may also be burdened with constraints such as old hardware and system architectures which may not be adequate to run Ada. Once again, we must recognize the problems and plan accordingly.

Multiple phase contracts do not invoke recurring cost to initiate a new contract each phase and they promote a smoother flow and continuity from one phase of the development to another. With multiple phase contracts the contractor may be more prone to produce better documentation and plans, knowing that he will have to implement them in the next phase. On the other hand, if the documentation is not required at a specific milestone in the Statement of Work, the contractor may delay documentation until later, knowing he has the next phase to complete and deliver it. To keep this from happening, multiple phase contracts must have built in milestones equivalent to smaller phases. These milestones provide visibility of the contractor's progress and the appropriate degree of control. In turn, the Government must provide the support and monitoring necessary to verify that the contractor has met the established milestones and may progress to the next phase.

Single phase contracts seem to be easier to control because we do not have to worry about the contractor advancing to the next phase before he has the appropriate documentation and planning. However, if the contract is for a single phase, such as Conceptual Phase, the winner of the Full Scale Engineering Development phase contract will have to inherit the technical documents produced by the first contractor. Early planning documents such as the Software Development Plan, Software Quality Assurance Plan, and Software Configuration Management Plan which could have been written in the Conceptual Phase are delayed until the beginning of the Full Scale Engineering Development Phase. In some cases the effort may progress along for several months without an approved Software Development Plan, Software Quality Assurance Plan, or Software Configuration Management Plan. One way of handling this problem is to have multiple contractors for the earlier Conceptual Phase. Require each contractor generate and deliver a Software Development Plan, Software Quality Assurance Plan, and Software Configuration Management Plan during the Conceptual Phase. Limit Full Scale Engineering Development competition to the Conceptual Phase contractors. Require the winner to follow the Software Development Plan, Software Quality Assurance Plan, and Software Configuration Management Plan they generated as a deliverable of the Conceptual Phase.

With the advent of Ada the single phasing approach presents another problem. The System/Segment Specification is usually generated during the Conceptual phase either by a Government or a contractor system engineer. Traditionally, the software development phase is perceived to begin after the System/Segment Specification is generated; however, this is no longer the case. DOD-STD-2167 emphasizes that the software development cycle can take place during any phase of the weapons system lifecycle and may cross boundaries. The Ada software development cycle must begin concurrently with systems engineering, traditionally, in the Conceptual Phase. It is the responsibility of software engineering to ensure that the computing architecture is capable of accepting the Ada implementation. The Data Item Description associated with DOD-STD-2167, for a System/Segment Specification accommodates this observation. For a single Conceptual Phase contract, with the System/Segment

Specification as the final product, there is no way that the software engineer from the following Full Scale Engineering Development phase contract can have any input to the System/Segment Specification. That input is necessary to ensure the Ada architectural requirements are accommodated in the System/Segment Specification.

Statement of Work Writing

The Statement of Work purveys to the contractor the tasks he is required to perform; the Contract Data Requirements List orders the documentation and software which result from these tasks. All too often the Statement of Work invokes blanket standards and requirements which are unrealistic, conflicting and unachievable. This will often, especially when DOD-STD-2167 is invoked, result in a conflicting and unachievable Statement of Work. For example, the requirements of our current software development standard, DOD-STD-2167, do not support the Ada standard, DOD-STD-1815A; however, according to current requirements, we must invoke both. When the Ada developer chooses an alternate approach or methodology, to the one specified in DOD-STD-2167, DOD-STD-2167 and its associated Data Item Descriptions require major tailoring, beyond simple deletion. Fortunately, DOD-STD-2167 advocates tailoring of itself.

A good contract Statement of Work decreases the risks to both the contractor and Government. One risk reduction method is to request that the contractor include proposed tailoring of DOD-STD-2167 to accommodate Ada in his bid package. This will hopefully minimize unrealistic, unnecessary, or impossible requirements. This will also provide the Government with insight into the contractor's experience and knowledge of Ada and his ability to apply that knowledge to the Government's requirements.

Statement of Work preparation requires the combined efforts of many people in specialized areas. Many times the individuals looking at the Statement of Work in these areas are hardware oriented and not software knowledgeable. When this situation occurs, the software is not properly addressed in the Statement of Work and may not be visible. A team of hardware and software professionals is required to ensure that the technical tasks, the legal aspects, and the engineering disciplines are adequately addressed.

Technical Perspective. From the technical perspective, it is necessary for a software engineer to have input to the Statement of Work to ensure Ada and Ada design techniques are properly addressed. An example of this would be the requirement for an object oriented design approach rather than a top-down design approach.

The contractor possesses past experience and a depth of knowledge that should be capitalized on. Therefore, it is necessary to give contractors bidding on the contract an opportunity to have inputs to the contractual requirements. This can be accomplished through "Instructions to Bidder". The Instructions to Bidders can request that the bidder supply a draft Software Development Plan and tailored Data Item Descriptions with his bid proposal. The Software Development Plan and Data Item Descriptions should be in accordance with DOD-STD-2167, but tailored to support Ada. The Software Development Plan should include a Software Configuration Management Plan, Software Quality Assurance Plan, and Software Standards and Procedures Manual. The contactor should include the System/Segment Specification, if appropriate, the Software Top Level Design Document and the Software Detailed Design Document. The Software Development Plan and tailored Data Item Descriptions should reflect the contractor's proposed alternate software development cycle and thus provide contract proposal evaluators with insight into the contractor's understanding, ability, and experience in developing and documenting Ada systems. It will also provide the contractor the opportunity to impact the Statement of Work requirements and apply what they have already learned and are familiar with. The Statement of Work should require delivery of the final Software Development Plan within 30 days of contract award and the contractor must be required to follow the Software Development Plan.

Legal Perspective. When writing a contract Statement of Work, we must accommodate not only the technical requirements but also the management and legal requirements. From the legal aspect we must always determine and levy the appropriate data and software rights. Standard clauses are placed in the contract to ensure that the proper rights are required. With Ada the issue of software rights and full design disclosure becomes a complex legal issue that must be addressed in the Statement of Work.

The objective of Ada is to create reusable software code. This encompasses creating and utilizing a "software library" of reusable code. To exercise the reusability factor across developers, we must have rights and access to the software library. However, this may not be as simple and straightforward as it sounds.

Software development contractors may not be willing to sell the rights or visibility to their Ada software libraries. Ada libraries can inherently contain proprietary factors which give the contractor a competitive edge. However, without the rights to the Ada library it will be difficult for the Government to maintain a competitive reprocurement posture which is mandated by the Department of Defense. Without knowledge of, and rights to, the Ada library we will not have full design disclosure and, for the Government, the real objective and benefit of Ada cannot be achieved. The Ada software will be reusable by the developing contractor only. Without knowledge and visibility of the Ada software library maintenance of the operational software becomes extremely difficult, if not impossible. A decision must be made whether to require full rights and design disclosure to the Ada library in the contract or treat it as a "black box" with each successive contractor utilizing or developing his own library to meet the need.

Engineering Discipline Perspective. In the past we realized that hardware efforts required an engineering, disciplined approach. We started invoking and following a structured phased approach and required configuration management programs and quality assurance programs. Unfortunately, this was not the case with software. Software development was sometimes looked upon and treated as a "black magic". Good engineering discipline, such as a structured, phased approach, Software Configuration Management and Software Quality Assurance, were not applied. Therefore, the word "software" became synonymous with the words "over cost", "over schedule", and "under documented." We have since learned that all software development efforts require an engineering, disciplined approach. Ada provides a structure and methodology which supports this approach. The Statement of Work must be written to require a phased approach, a Software Configuration Management Program and a Software Quality Assurance Program which supports and utilizes Ada's inherent capabilities.

Proposal Evaluation

The SDP. The Software Development Plan submitted with the contractor's proposal provides the information necessary to evaluate the contractor's proposed software development effort. The Software Development Plan is a keystone document in any software acquisition and development, but it is particularly important and informative in an Ada effort. For example, if the contractor really has previous Ada experience, he should have an Ada software library with reusable code in it. The Software Development Plan should specify how much existing Ada code the contractor anticipates using and how much of the new code he generates will be reusable?

The Software Development Plan not only provides information on the contractor's software development resources, Software Configuration Management Program, Software Quality Assurance Program, and proposed alternate software development approach, it also provides insight into management support and understanding of the nuances unique to an Ada software development. Without management's understanding of Ada, planning for Ada, and committment to Ada the end goal of decreased lifecycle cost cannot be achieved. When money and time get short, the first inclination is to cut corners in training, resource allocation, schedules, and documentation. This could make Ada objectives unachievable. The Software Development Plan should present schedules, design techniques, manpower loading, training activities, etc., which correlate with the use of Ada.

Management planning must anticipate and deal with the risks an Ada system development carries with it. Our Ada experience to-date indicates that much of our non-Ada historic data is not valid with an Ada software development. For example, in the past if a program needed to be designed and generated in a certain language unknown to any of the designers, it was preferable to choose a designer experienced in another language and put him to work on the job. With Ada this does not appear to be true. Since designers in Fortran, Cobol, Assembly, etc., use functional thinking and functional decomposition, it is sometimes difficult for them to switch to something as abstract as Ada and Object Oriented Design. Quite often, when an experienced Fortran system designer tries to design an Ada system without adequate training, they end up with a poor Fortran and Ada design, or what we call Adatran. It may be easier to take a novice and teach him Ada techniques than to try to retrain an experienced functional designer in Ada. Additionally, there is evidence which shows that the Ada learning curve for a Fortran or Cobol programmer may be longer than the learning curve for a programmer in a structured language like Pascal.[2]

Productivity rates are another important aspect for management to consider and plan for. Productivity is a difficult thing to gage, however, it appears that it requires experience on a minimum of three Ada developments before a programmer's productivity with Ada is equal to his productivity using other languages. Reduced productivity could cause major schedule slippage and overspending. How many experienced Ada programmers does the contractor have? What is the extent of their experience? Management must be aware of all this and plan accordingly.

The proposed schedule should be another good indicator of management's understanding and experience in the Ada software development arena. Compared to a non-Ada software development, an Ada schedule should indicate increased expenditures and activity during the early phases of development and less during the coding phase.

The alternative software development approach is an important part of the Software Development Plan. However, a bidder proposing an alternative software development cycle for an Ada project is faced with a dilemma. On one hand he has the desire to do the job in a timely and cost effective manner and on the other hand he is faced with the absence of any objective criteria for judging an alternative software development cycle. The proposal evaluator is faced with a similar problem. He can no longer use the cookbook approach to evaluate the software development cycle and he too is faced with the absence of any objective criteria for judging an alternative software development cycle. Therefore, we have two problems. The first problem is how to evaluate an alternate software development cycle. The second problem is understanding how Ada affects the cycle.

A simplified information model is a tool that can be used to help solve these problems. The software development cycle is a simplified information model of a complex process. There are six elements to consider in establishing criteria for using a simplified information model to evaluate alternative software development models. We will first cite the elements of a simplified information model[3] that are critical to the software development cycle. To create a simplified model we use simplifying assumptions based upon experience and we ignore some relationships by not including them in the model. The use of a simplified model is based on the assumption that a manager will make any day to day decisions based on heuristic reasoning and that the formal reasoning implied in the model will be used periodically to make programmatic decisions. Initial planning information is derived from the model as budgets and schedules. After project initiation, the information received from the model or from other sources is control information and is used to modify the model or modify current decisions.

(1) Simplifying Assumptions. In order to reduce the complexity of the software development cycle we make assumptions that simplify some aspect of the model. An example of a correct simplifying assumption is assuming that each Ada programmer produces code at a rate equal to some average rate. While this is not true, it simplifies the model. An example of an incorrect simplifying assumption is assuming that non-Ada programmers will be equally productive on their first Ada project. The simplification is that Ada is just another programming language.

(2) Ignored Relationships. In order to reduce the complexity of developing a software development cycle we must ignore certain relationships. For example, it would be correct to ignore the relationship between a particular manager and the software development cycle. While a manager may be identified for a project, we cannot guarantee that he will be the manager throughout the entire software development cycle. It would be incorrect to ignore the relationship between a programmer's Ada experience and the programmer's productivity.

(3) Heuristic Reasoning. Heuristic reasoning is characterized by the use of "rules of thumb" and is the province of humans and expert systems. It should not be part of the software development cycle. An example of incorrectly including heuristic reasoning is the statement, "All problems encountered during the software design review will be analyzed and corrected by the programmer." This statement is incorrect because it assumes that all software design problems are caused by, and can be corrected by the programmer. An example of correctly excluding heuristic reasoning is the statement, "All problems encountered during the software design review will be analyzed and appropriate corrective action taken." This statement does not specify what the corrective action will be and is correct because it is impossible to state what the problems, their causes, and their solutions will be in the future.

(4) Formal Reasoning. Formal reasoning is the "stuff" that software development cycles are made of. It provides the answers to who, what, when, how, and when necessary, where and why. The formal reasoning must not only be correct but it must also be complete. An example of correct and complete formal reasoning is the statement, "Following the successful compilation of a package identified for Clean Room Testing,[4] Software Development will graduate the package from the development library to the test library. The package is then under the formal control of Software Configuration Management." This statement of formal reasoning must include the answer to the question "why". It is not obvious that one form of Clean Room Testing is to submit code for formal testing after compilation and informal testing by Software Development. By modifying the same example we can easily create an incorrect and incomplete example. "Following the completion of a package it is given to Software Configuration Management for Clean Room Testing."

(5) Planning Information. There are two categories of planning information: static and dynamic Static planning information is the software development cycle itself. It is static at any point in time and any product can be identified as being at some point in a process controlled by the software development cycle. Dynamic planning information is the information that flows directly out of the software development cycle once it has been implemented. This information is analyzed and used to change the software development cycle itself. An example of dynamic planning information is the statement, "Software Engineering will conduct a semiannual audit of software development practices, analyze the results and make appropriate changes to the Software Development Plan as needed." It shows that management has included the ability to monitor, evaluate and correct the software development cycle.

(6) Control Information. Information that the software development cycle produces about the product is control information. It provides management with the information needed to determine whether or not to intervene in the process. An example of control information can be found in the statement, "Product quality information will be collected and analyzed. Recommendations for corrective action will be provided." (It should be noted that the analysis of the data may reveal a need to change the software development cycle, i.e., it is a planning problem, and/or a need to change the product.)

The Ada Impact. We now have established the criteria to be used in judging any software development cycle independent of a standard. Before we can use the criteria in an Ada context, we must understand what sets Ada apart from other programming languages. More importantly, we must establish a criteria to distinguish "good" Ada developments from "Ada-tran". A topological approach can be used to explain the differences in each generation of programming languages.

> Since assembly languages are still commonly used for embedded computer systems, we have provided their topology...it does point out the fact that assembly languages provide no inherent structure...Certainly, they provide great flexibility in creating systems, and it is possible to write structured assembly code. However, once a solution reaches even a moderate size, the nature of the assembly complicates it.[5]

We take exception to one minor, but significant point. If we view the assembly code from the point of view of a programmer who has implemented some algorithm, say a function to compute SINE(X)/X, there is no structure visible in the assembly code. As anyone who has ever inherited assembly code will agree, it is possible to understand exactly what each line of code does without ever being able to understand what the total program does or why. This is because the algorithm is not readily visible in the assembly language implementation. But if we view the assembly code from the point of view of the processor there is a structure. The assembly code is a perfect mapping from a line of assembly language onto the processor micro-code. It is this fact that allows for the extreme flexibility in controlling the processor. While this may seem obvious, since that is what assembly language is designed to do, the significance of the concept will be illustrated shortly.

> Ada was developed at the end of the language "generation gap" and so has been influenced by contemporary software methods. In a sense, it is the first of a new generation of languages...Ada's topology is not flat like those of the previous generations [first, second and third], but rather is three-dimensional...this topology helps us to localize design decisions and so the structure of the original design is more easily maintained as modifications are made.[6]

From the point of view of the programmer who has implmented the algorithm to compute SINE(X)/X in Ada, this three-dimensional structure has no obvious advantage. In fact, Ada would probably seem very restrictive compared to Fortran. From this point of view, Ada is just another programming language that represents an algorithm in a human recognizable form.

If we view Ada from the point of view of the total system, much as we viewed the assembly language from the point of view of the processor, we again discover the possibility for a perfect mapping from Ada onto the system. Just as the mapping of assembly language onto the processor micro-code gives us flexibility in controlling a processor, the mapping of Ada onto the system gives us flexibility in controlling an evolving system. This control is the direct result of correctly utilizing the levels of abstraction inherent in Ada and separates a good Ada development from "Ada-tran".

Now that we have addressed the Ada impacts as well as the software development cycle evaluation criteria, we need to look once more at how they fit together in the software development cycle. We have observed that most software development cycles begin with inputs from systems engineering (i.e., system specification) and end when the software is input to system integration. This traditional ending is acceptable if we assume that system integration marks the beginning of a new software development cycle. However, we disagree with the idea that an Ada software development cycle begins when systems engineering provides its inputs. The Ada software development cycle must begin concurrently with systems engineering. This is due to the fact that the computing architecture must be capable of accepting the Ada implementation.

It has been stated that the software engineer is the systems engineer of last resort because the people who are performing systems engineering functions are actually subject matter experts.[7] Our own experience has shown that true systems engineers often have neither the time nor the inclination to become experts in Ada and its design techniques. While such a person would be extremely valuable, we believe it is counter-productive in any organization producing software intensive weapons systems. The rationale for this belief is that modern industrialization is based on the division of labor into specialities. The "Jack-of-all-trades" is a product of a cottage industry. We passed out of the software cottage industry stage when engineers began to use programmers. The professional programmers marked the beginning of the software industrial age. Software Engineering and its language, Ada, marks the beginning of the software factory age. It has already been reported that the concepts we argue for have produced interchangeable programmers.[8] When our tools are complete and our reusable component libraries are full, the software factory age will truly be here. For this reason we believe it anachronistic to expect systems engineers to become expert in mapping Ada solutions onto systems. Our experience shows that the software development cycle must begin with the systems engineering effort. The failure or inability to do so produces undesirable results which we discussed earlier.

Pre-Award Survey. After the contractor's proposal has been reviewed a team of evaluators should visit each bidder's facility and perform a pre-award survey. A pre-award survey is performed to determine which contractors actually have the resources and capacity to do the job. The team then verifies that these contractors have followed the procedures and approach specified in Software Development Plans on past and current projects. The team should be presented with proof of these factors. The contractor should make available software documentation, status accounting reports, meeting minutes, or other information from past and present projects which corroborate his proposal, the draft Software Development Plan, and tailored Data Item Descriptions. The results of the pre-award survey should be factored into the final proposal evaluation.

Conclusion

This paper is a snapshot in time which identifies areas where Ada increases procurement risks and our approach to reducing them. We feel a growing confidence in our ability to successfully reduce such risks in Ada procurement. This confidence arises from the sharing of common Ada problems and solutions among Government and industry organizations. This paper has been our attempt to add to that body of shared experience.

1. "No Silver Bullet", by Frederick P. Brooks, Jr., Computer Magazine, April 1987.

2. Dr. Randy Jensen, Ada Jovial Users Group, 19 Nov. 87.

3. Joel S. Demski, Information Analysis, Adison-Wesley Publishing Company, Reading, Massachusetts, 1972.

4. Harlan D. Mills, et al, "Cleanroom Software Engineering", IEEE Software, September, 1987.

5. Grady Booch, Software Engineering With Ada, Second Edition, The Benjamin/Cummings Publishing Company, Inc., 1986, page 40.

6. Ibid page 41.

7. Mark S. Gerhardt, "Don't Blame Ada", Defense Science & Electronics, August, 1987, page 53.

8. David Harvey, "Ada Field Development", Defense Science & Electronics, February 1987, page 27.

Sandra A. Pryor is a software engineer in the Weapons Systems Software Engineering Branch at the Naval Weapons Center. She supports various weapons systems programs in the areas of software program management and software program planning and acquisition. She is the software representative on the Naval Weapons Center Acquisition Requirements Data Requirements Review Board and teaches courses in software standards and specifications and software development emphasizing configuration management. Her technical interests include the software engineering disciplines, weapons software acquisition strategy, and technical program management. Correspondence should be addressed to Commander, Naval Weapons Center (Code 3627), China Lake, CA 93555-6001.

George J. Smith is a computer scientist in the Weapons Systems Software Engineering Branch at the Naval Weapons Center. His professional interests include Software Engineering Management, Software Life Cycle Management and Software Architecture. Smith received a BS in economics from St. John's University, a BA in computer science from San Diego State University, and is currently a candidate for an MS in computer science from California State University, Chico. He is a member of ACM and IEEE Computer Society. Correspondence should be addressed to Commander, Naval Weapons Center (Code 3627), China Lake, CA 93555-6001.

MINIMIZING ADA RISKS THROUGH BENCHMARKING

Christine L. Braun

SofTech, Inc.

Abstract

DoD's mandate requires the use of Ada for most new developments, forcing many to use it for the first time. Ada, like any new technology, poses risks. These risks can be controlled and minimized through proper planning before key decisions are made. In particular, an important part of such planning is benchmarking. Benchmarking attempts to replicate key critical aspects of the project and to measure the effectiveness of proposed approaches. It guides the choice of Ada compiler, methods, and tools. It can also influence design decisions and hardware selection. It allows program managers to be "informed consumers", and to undertake Ada projects with a high degree of confidence that there will be no surprises.

WHY "ADA RISKS"?

Risk is not a new problem in DoD systems. Most DoD system development programs push the state of the technology in one or more areas. New processors and equipment are used, existing ones are used in new ways, new algorithms are developed, and ever-more-stringent performance demands are met. We are all used to working this way, and have developed an understanding of how to approach these risks. Basically, we try out new ideas first, before committing absolutely to them. Our management understands this -- major new business developments and proposal efforts are accompanied by significant trade studies devoted to just such experimentation. Customers understand this -- many key programs include initial prototyping or advance development phases.

Why haven't we applied this strategy to the software development aspect of these systems? Not because software development was perceived as easy or low risk, but because we did not have alternatives. Few choices were possible -- programming was programming. With no choices to be made, there was little purpose to such experimentation.

First, the software development approach was usually not new. Algorithms might be new, but methods, language, and tools were typically those the programmers had used before. Much software development was in assembly language. Even when the machine was new, the new assembly language was "just another language". The programmer could use methods and design approaches he had used before. Because assembly language is low-level, it did not constrain these higher-level choices. Second, the software development approach has not been perceived as a driver of other system decisions. For example, one processor would not be selected over another because it has a simpler, more reliable, or more powerful assembly language. All assembly languages work, and programmers were expected to be able to work with what they were given. Finally, because software development was always done in much the same way, we had developed (reasonably) effective metrics for estimating software schedule, size, and performance.

Of course, this complacency has not always been warranted. Consider the efforts made by the Air Force to transition to JOVIAL J73. They faced many of the same problems of newness that Ada poses, and were hurt badly by them. Programming in J73 was in fact different. Programmer unfamiliarity with the language, functional limitations on what the language supported, immaturity of compilers and tools, and lack of reliable estimating procedures caused major schedule, cost, and performance problems, to the point where some programs scrapped J73 and reverted to assembly language. J73, in fact, was not "just another language".

How can we hope to be more successful with Ada? Not satisfied with the many "newnesses" resulting simply from a new language, we are also demanding a revolution in software engineering methodologies -- compilable PDLs, object-oriented design, software reusability, and all sorts of other fancy new ideas are being bandied about. All this new technology means new choices -- various compilers and tools, methodologies, and design approaches are possible. This is a long way from the world where the programmer just combined his knowledge and his assembly language skills to solve the problem. In fact, the software development approach is now a key decision area and a key risk area. We must treat this risk like we have treated traditional risks -- we must understand it, consider alternatives, and conduct experiments to determine the best approach. In other words, we must <u>benchmark</u>.

THE BENCHMARKING PROCESS

Benchmarking, as we use the term, means experimentation to select the best among available alternative approaches. It may be an experiment specifically designed to evaluate more than one approach, or may be designed to prove the expected effectiveness of a single preferred approach. In either case, it allows us to proceed with development with a greatly-improved degree of confidence in our approach. It is a simple concept -- <u>try it out before you commit to it</u>.

The term "benchmarking" has most often been used in conjunction with performance measurement. While that is an important aspect for evaluation, we use the term more broadly. Benchmarking also allows us to evaluate the effectiveness of a proposed approach in meeting system functional requirements, in providing a maintainable product, and in conforming to project cost and schedule constraints.

The benchmarking process involves five major steps. These are described in the following paragraphs.

<u>Identify key risk areas for evaluation</u>. These should be aspects of the software development that are sources of potential risk. Risk typically arises when something new is being attempted -- for example, when a tool or method is to be used for the first time. Risk also arises from difficult-to-meet system requirements, either for performance or functionality. Difficult cost and schedule constraints are another source of risk. Risks should be fairly specific -- e.g., "using Ada for the first time" is too general a statement to be useful. Risks should be stated in terms of what might go wrong -- e.g., "lack of familiarity with Ada may make it impossible for our programmers to meet the schedules we have set". The next section of this paper discusses a number of possible Ada risk areas.

<u>Identify available alternatives for each area</u>. Ideally, each risk area listed should have alternative approaches. (Hopefully, these are less extreme than "go back to assembly language" or "give up the whole idea".) Even if you are fairly confident that you know what the best approach is, devote some time to identifying other possibilities.

<u>Devise an experiment for each key risk area</u>. Describe how you will either

a) demonstrate that your planned approach will in fact work, or

b) evaluate and select among the various alternatives, <u>and</u> demonstrate that the chosen one will work

It's not enough to find a way to try out the planned or alternative approaches. You must determine how you will gather results from the experiment, and identify the results that must be achieved to demonstrate that an approach will work. Remember that your goal is to find, not the <u>best</u> approach, but a <u>sufficient</u> approach -- one that will work. An approach can be sufficient without being best (and you might want to choose it for another reason, e.g. cost); conversely, even the best approach may not be sufficient.

Note that typically experiments for different risk areas can and should be combined. For example, the same single experiment might address object code performance and size requirements.

Conduct experiments. Carry out the experiments and document the results.

Select a suitable approach for each risk area. If the experiments were devised correctly, each will have resulted in an evaluation of suitability for each candidate approach. At this point, you must select the overall set of approaches that best meets your requirements. Typically approaches will interact -- for example, the compiler that provides the best object code execution speed may be the least robust. The goal is to select an overall set of approaches sufficient to do the job.

Perhaps an experiment for a risk area might determine that none of the candidate approaches is sufficient to meet requirements. If this occurs, several responses are possible:

a) Relax the requirements.

b) Identify another alternative approach that would suffice.

c) Apply extra resources and management attention to the area at risk.

For example, suppose benchmarking shows that no compiler being evaluated appears likely to result in sufficient execution speed. Perhaps the performance requirements can be relaxed. Alternative approaches include using another compiler, changing to a faster processor, or coding performance-critical segments in assembly language. If no changes or alternatives are possible, top programmers can be assigned and extra effort devoted to performance monitoring and tuning. Benchmarking provides a sound basis for justifying these changes or extra efforts. (It also helps develop a realistic cost estimate for doing the job -- a concern of both customer and contractor.)

BENCHMARKING KEY RISK AREAS

In this section, we will examine some of the key risk areas that might be considered when using Ada, and briefly discuss some approaches to benchmarking them.

Compiler Maturity

The entire software development activity is dependent on the Ada compiler. It is critical that the compiler work -- that it be bug-free and that it compile Ada correctly. Of course, the first step in assuring this is to use a validated compiler. However, the validation process addresses correctness rather than robustness -- a compiler can pass validation tests and still be highly unreliable. A good approach to this problem is to try to gather information on practical experience of other projects that have used the intended compiler, a more realistic possibility now that Ada is seeing wider use. However, the compiler under consideration may be too new for this to be possible. Moreover, some efforts are obliged to depend on a compiler that is still in development, typically because no validated compiler exists for the chosen target computer. (This might be a reason to consider another computer, by the way.)

In cases where there is not adequate practical experience to establish confidence in the compiler's maturity, it is essential to do some independent benchmarking. The goal here should not (and realistically cannot) be to duplicate the correctness testing done by the validation process. Instead, the compiler should be tested on the kinds of programs likely to stress it during the project. This can be done by developing one or more benchmark programs designed to exercise the compiler realistically. The characteristics of a good robustness benchmark include:

a) size -- The program(s) should be fairly large. The validation tests are small and test single features. Many validated compilers fall down on large, multi-feature programs.

b) representative use of language features -- The program(s) should use the Ada features the project will use. For example, if you plan to use tasking extensively, exercise it well. If you depend on generics, try them out in a variety of ways.

c) presence of programming errors -- Many compilers handle correct programs fine, but fall apart on incorrect programs. You do not want a compiler that bombs out when it encounters a syntax error; it should diagnose it properly and continue processing. Various kinds of likely programming errors should be included.

It may be that all compilers tested will have some problem meeting these tests. In this case, consider the possibility of workarounds. (The compiler vendor may be able to recommend these.) Be sure to try them out, and to document them for the programming staff.

Compiler Capability

As noted above, compiler validation tests that a compiler processes the Ada language correctly. However, many variations are still possible. These include:

a) optional features -- The Ada language definition includes a number of features that are optional; a compiler can be validated without supporting them. An important example of this is "**pragma interface**", a capability that allows the Ada program to interface with programs in another language. Furthermore, this feature can be implemented in a variety of ways; for example, only very primitive calls may be supported. If your application requires such interfaces, you must be sure the compiler provides the required support.

b) runtime support -- Many Ada language features are implemented by compiler-generated calls to a runtime support library provided by the compiler vendor. These include task and memory management. A variety of algorithms can be used to implement these features, and these can make a difference to the function and performance of an application. For example, the task management algorithms can influence the order in which tasks are scheduled. As another example, a memory management algorithm greatly influences performance in its choice of strategy for reclaiming freed storage. It is important to be sure that the runtime algorithms provided with the compiler meet your application needs.

c) capacities -- Many compilers have built-in capacity limitations of various sorts. For example, a compiler might limit the number of tasks that can be defined in a single program, or the number that can be active concurrently. It might limit the size of the complete Ada program, the number of modules that can be linked together, the number of objects declared, the number of generics, or many other similar things. It is essential to be aware of any capacity restrictions that might influence your ability to implement the system.

The way to address these questions is to (1) list the areas in which variations among compilers might exist, (2) for each area, identify any dependencies you might have, and (3) define a means for determining whether candidate compilers meet your dependencies. The starting point is a thorough understanding of what options the language allows. This requires someone with a pretty strong knowledge of the Ada Language Reference Manual. It is then necessary to understand the implications of these choices on your application. This can be difficult, because the implications are low-level and complex. A SIGADA Working Group, the Ada Run-Time Environment Working Group (ARTEWG), is studying some of these issues, particularly as they apply to the implementation of embedded real-time systems. A familiarity with ARTEWG's work will help understand implications for your project.

It might be possible to address these questions adequately without benchmarking per se. Ideally, the vendor will document the choices and limitations incorporated in his product.

However, this is not always the case, particularly with capacities, so some testing may be required simply to discover these compiler properties. Also, knowing the compiler properties is not always enough. In order to match them to your application needs, some benchmarking may be needed. For example, if there is a limitation of task nesting depth at runtime, it may be necessary to develop some kind of simulation to determine what nesting depth your application will require.

This is an area in which, if analysis or benchmarking do not identify any compiler that meets project needs, other recourse may be possible. Many compiler vendors are prepared to add optional features, or to tailor runtime environments, to particular customer requirements. Because matching application requirements and runtime support needs is complex, some vendors will work with application vendors to design specific customizations that best meet application needs.

Object Code Performance

Execution speed is an important concern in all applications; in some it is critical. This is probably the greatest area of concern for those adopting Ada for the first time. It is a valid concern -- high-level languages generally give somewhat worse performance than assembly language, particularly when the language is new and compiler technology for the language not fully developed. However, this is an area where a lot can be done.

First, it is important to consider performance up front. Performance is a true requirement, just like functionality. If performance is critical to project success, there is really no point in undertaking the project without some degree of analysis to establish confidence that requirements can be met. This is no different from setting out on a major development without satisfying yourself that the key algorithms work -- it just doesn't make sense.

Second, it is important to select a total solution that can meet performance requirements. This total solution might involve:

a) choosing a compiler that has the best possible object performance for the intended target computer

b) choosing a target computer that has a compiler that can meet performance requirements

c) adding processing capacity to the chosen target computer

d) making design decisions that optimize performance

e) identifying critical portions of the system that may be candidates for implementation in assembly language

f) obtaining additional application-specific optimizations or runtime support library tailoring from the compiler vendor

It is vital to consider this total solution -- to remember that compiler choice is not the only factor.

How do we approach performance benchmarking? It is relatively easy to measure performance on some benchmark test -- the hard part is establishing confidence that the benchmark is representative of the application. This is a subject that has been treated extensively elsewhere. Simulations and loading models can be used. For Ada, it is particularly important that any such approach take into the account the tasking behavior of the application. This is an area of substantial possible execution overhead, and one where significant tuning and customization are possible. Another Ada consideration is, if assembly language insertions are required, to obtain a compiler that supports them.

Object Code Size

This issue is very similar to the performance issue. It deals with how much memory the object code takes up, rather than how fast it runs. In some applications this may not be very important. In embedded applications, where size and weight limitations apply, it can be critical. The computer system and its resident software must fit in a missile, an airplane cockpit, or whatever. If an additional memory bank is needed at the last minute, the system may simply not fit. Just as there is no point in undertaking a project without confidence the system will run fast enough, there is not point in starting without being sure the software will fit in its intended home. (A recent DoD Ada development effort, which did not do advance size benchmarking, ran into exactly this problem. The embedded computer had to be replaced with an entirely different one after implementation was almost complete, because there was simply no way to make the system fit with the first approach. Cost and schedule impact was significant.)

Like performance, this issue requires a total system approach. The same aspects listed above apply here. This is an area where a hardware solution may be particularly appropriate.

Design Feasibility with Ada

There is an interplay between chosen design solutions and the implementation language, Ada. Some kinds of design choices may be difficult or even impossible to implement in Ada -- this is particularly true of designs developed by people who have always used assembly language before. Some choices may have major performance implications, as indicated above. This is particularly true in the realtime area. Some choices may be entirely feasible in Ada, but may not be "good Ada". This means that they may not take advantage of Ada's benefits. They may not have the reliability, maintainability, portability, etc. characteristics that Ada offers. This typically occurs when the programmer avoids all Ada features that do not closely resemble those in the language(s) he is used to.

Performance implications of design choices would probably be dealt with in the same set of experiments used to address the performance issue in general, as noted above. It helps to obtain recommendations from the compiler vendor as to relative efficiency of various language constructs in his implementation; many offer such data.

Benchmarking may be useful in determining whether certain design choices are feasible in Ada. This is particularly true with first-time Ada users. For example, the designer of the software that processes interrupts from external system equipment might be unsure that his planned design will work, perhaps because he is not sure how Ada, and his intended compiler, handle interrupts. It is straightforward to develop small benchmarks to try out such designs. This is especially important when the design choice might influence overall system design.

The problem of "good Ada" may not be best addressed by benchmarking. It is probably not feasible to try to measure the maintainablility of alternative designs. However, if particular design constructs appear many times in the system, it may be worthwhile to develop really good model Ada solutions to promulgate to the programming staff. With a staff of new Ada users, a project guide documenting various such Ada usage guidelines specific to the application can be valuable. Special training in these may also be appropriate.

Portability Potential

Portability is a widely-touted Ada benefit. Some new system development efforts are planning to take advantage of that benefit. Developers may plan to develop initially on one computer, then port to another prior to delivery. They may plan to use the same code on different processors within the system. They may plan to reuse code from another application developed on another computer. All of these are good ideas. However, portability cannot be taken for granted, even with Ada. An Ada program not specifically programmed to be portable will almost certainly not be. There are several possible reasons for this:

a) Some Ada features (by intent) depend on target machine characteristics. An example is numeric representations, which influence precision of results. Clearly, machine-dependent features such as representation specifications are not portable.

b) The operation and performance of the software may depend on specific char^cteristics of the runtime support library, as discussed. Porting the software can change its behavior.

c) If optional language features (e.g. pragmas) are used, they may not be available on the new machine.

d) Software may depend on characteristics of external hardware, e.g. user terminals, that may differ for the new machine.

e) Software may interface to other programs, e.g. a database management system, that are not available on the new machine.

The way to approach portability is to (1) identify any specific portability requirements you have, (2) obtain and use a good set of portability guidelines, and (3) consider benchmarking as a proof of principle. In this case, benchmarking involves trying out your approach to portability in each of the above areas that apply. For example, if you have a dependency on numeric representations and have used Ada data typing to handle this, try out your strategy by actually porting some representative code and comparing execution results. If you plan to handle a DBMS dependency by the use of a standard interface such as SQL, write a test program that exercises the DBMS calls, and verify that it runs identically on the two machines. As the project progresses, don't assume that you've taken care of any portability risks. If possible, start porting early, proceeding incrementally as development progresses. This allows early identification of any unanticipated problems.

Reusability Potential

Reusability is handled much like portability. Like portability, it is an Ada benefit that offers attractive advantages, but does not come free. Reusability refers to the ability to use a piece of applications code in a new application, whereas portability refers to the ability to run it on a new machine. (Reusability, if the two applications run on different machines, has portability as an underlying requirement.) Reusability deals with software interfaces, while portability deals with hardware interfaces.

Reusability is harder to specify precisely than portability, and generally more difficult to achieve. We stated above that any desired portability properties should be specifically stated as system requirements. The same is true of reusability, but requirements are harder to specify. Existing code not originally designed to be reusable is rarely so without extensive modification -- interfaces and exact functionality are too unlikely to be compatible. Components can be designed for reuse, generally through the use of parameterization capabilities such as Ada generics, but the reuse is still limited to "reusers" who can match the interface provided. Only when the programmer is specifically tasked to write a reusable component, provided with guidelines for doing so, and working to a fully-defined interface that will be adopted by reusers, can reuse be depended on.

Benchmarking can be used to help define the interface to be provided by a reusable component. A "dummy" component providing the anticipated interface can be developed and tested with sample fragments from the reusing programs. This helps ensure that the interface is adequately generalized.

Environment Effectiveness

Usually, when undertaking a first Ada project, programmers are confronted with a completely new development environment. The compiler is certainly new. There are usually other language processing tools (e.g. linker, loader) directly associated with the compiler. Other special-purpose tools may also have been acquired from the compiler vendor or other sources. These can include configuration management tools, design aids, documentation tools, PDL processors, etc. The programmers assigned to the new project may have used none of them before.

The key point of environment benchmarking is simple -- try out all these tools before handing them to the project team. Tool selection is one of the areas in which we have many options, so benchmarking has an obvious payoff. There are three objectives to environment benchmarking:

1) Determine that tools work as advertised.

2) Determine that they work together.

3) Observe the effects of their use.

Testing to see if tools work is straightforward. The important is to define what you want the tools to do for your project. (Perhaps you'll find out that the tool was never even intended to do that.) Then you test them to be sure they do so.

Determining if tools work together is much more complicated, assuming that tools did not all come together from the same vendor. The incompatibilities may be straightforward and obvious, such as incompatible file formats. Such incompatibilities are generally fixable. Incompatibilities may also be non-obvious and perhaps subjective. For example, a top-level design tool and a detailed design tool might have different ideas about where the boundary between those two levels lies. As part of the benchmarking process, you will need to interface the selected tools, or at least define clearly how this will be done.

Observing the effect of tool use includes some measurement of how the tool helps or hinders the developer. Included in this judgement is the critical issue of tool efficiency. Some Ada compilers execute very slowly, which can impact development cost and schedule. Tools that are hard to use, buggy, or have poor error recovery should also be discovered now. The quality of tool documentation, and the possible need for training in tool use, should also be considered. Tool benchmarking, to permit these observations, should be done by individuals representative of those who will be assigned to the project.

Methodology Effectiveness

Just as a first-time Ada project typically means a new set of tools, it also usually means new methodologies. Ada is perceived as a way of improving our overall approach to software engineering; consequently customers and managers typically demand that we use various new methodologies. These include new techniques for analysis and design, and well as new software life-cycle models. (For example, iterative development approaches such as rapid prototyping are popular.) New standards, e.g. DOD-STD-2167, are imposed.

Just like tools, we must try out methodologies before putting them in use. We have the same three goals -- see that they work, see that they work together, and observe their impact. This is a bit more difficult with methodologies than with tools because it is harder to define just what it means for a methodology to work. Also, many of their good effects are not apparent immediately, but evolve with experience with the methodology. It is essential, however, that we establish some definition of what we expect a methodology to do, and verify that it does that. This should be in concrete terms -- i.e. produce a particular kind of diagram that allows one to proceed to some subsequent activity.

The problem of methodologies working together is also significant. Methodologies have an unfortunate tendency to try to solve all the problems of software engineering, rather than sticking to one. Thus, when we try to adopt more than one we usually find that they overlap. We must clearly establish just what we intend to do with each, and demonstrate that they fit together in a deterministic ordered sequence. It is not appropriate to offer the project team a shopping list of methodologies that may or may not address their problems.

Observing the effect of methodology use, as noted above, is difficult. However, we can assess the ease of learning to use the methodologies, the quality of available user documentation, and the need for training. Here we will almost certainly determine areas where project-specific guidance is necessary to ensure that methodologies are used correctively, uniformly, and productively.

Schedule/Cost Estimate Reliability

Accurately estimating schedule and cost of software development has never been one of our strong points. Our only useful approach has been to gather empirical data on programmer productivity, etc. and base estimates on that data. With Ada and all the new technology surrounding it, we lose even that. There is much need in the industry to start gathering and promulgating such data, and this is beginning. However, what does a first-time Ada project do?

Using the early available Ada productivity data may be a mistake. Early figures vary widely, and in many cases seem to be produced by Ada evangelists (and hence to be positively slanted). Also, early figures are greatly skewed by programmer experience and training.

Benchmarking can help with this problem, but only through a fairly aggressive approach. If the only way to estimate accurately is by gathering empirical data, then we must use the benchmarking activity to do that. But because development cost depends on so many factors, the only way to gather really usable data is to essentially conduct a "practice" or "model" project.

This need not be as drastic as it sounds. A good example is the plan for the Army WWMCCS Information System (AWIS) project. This project concept specifies an initial phase devoted to (1) defining the overall set of approaches to all of the issues above, (2) applying them to a small piece of the application development, (3) refining and revising them as required to achieve maximum effectiveness, and (4) gathering productivity and cost data. This is an excellent approach. (However, it is necessary to keep in mind that the first phase must not just become another development project with attendant cost and schedule pressures. When this occurs, the first thing to go is any attempt to improve the approach.) Does such an approach really cost more or take longer? Not obviously, for don't we end up doing these things anyway? And typically later, when it costs more to change? However, this clearly requires the concurrence and support of our customer and management.

WHO PAYS FOR ALL THIS?

The preceding description makes benchmarking sound prohibitively expensive. It probably isn't as bad as it sounds. Most projects will not involve all of these decision areas. Also, as we noted at the beginning, these various analyses can typically be combined into some relatively small number of experiments. In the "model project" approach described above, all issues can be addressed in essentially a single experiment, while at the same time moving forward with the project. However, there is undeniably cost associated with the benchmarking process.

Is it additional cost, though? We would argue that it is not -- that in fact it will save money. Clearly we will have to try out all of our methods, tools, and approaches anyway; benchmarking just says we will do it up front, when bad decisions are less costly to reverse. We are, though, proposing a distinct activity in the project schedule, and hence a distinct cost item.

Who pays? There are a variety of approaches, but all fall into two basic categories -- the company or the customer.

Why would the company want to pay? A product company has no alternative. Such a company will invest in benchmarking if it thinks it will pay off in long-range cost savings. Because such a company will usually adopt a single set of tools and methodologies for all its software development, the payoff prospects are great. However, most of us are concerned with developing software on government contracts. As discussed initially, the question is analogous to deciding to invest in other trade studies, prototypes, etc. There are two reasons our management might want to invest in benchmarking:

1) because it provides a competitive edge in bidding and winning jobs

2) because it helps cost jobs accurately

Benchmarking can provide a competitive edge in several ways. It can potentially lead to some sort of early products or prototypes that we can actually demonstrate to the customer. It can show that we have a head start on getting the job done. It can give greater confidence that our approach will work. For management to believe this, it helps if the customer appreciates the problem. A customer who doesn't understand that there is risk involved won't appreciate the up-front work.

Clearly providing a better cost estimate is an advantage, although sometimes the news may not be what we want to hear. However, particularly in today's competitive enviroment, it's best to know what we're getting into.

Why would the customer want to pay? Because:

1) He wants to avoid risk.

2) He effectively pays now.

3) It will cost less in the long run.

4) He has more control in seeing that the benchmarking is done right when he pays.

Clearly, if the customer intends to support and control such benchmarking activities he must write this into his Statement of Work. Bidders will not be inclined to propose it otherwise. Also, when planning a comprehensive benchmarking activity such as the "model project" phase described above, the customer should provide some mechanism to allow results gathered to determine project cost and schedule estimates. Otherwise, a key benefit is lost.

Whatever structure we adopt to fit benchmarking into the software life cycle, it can bring us reduced risk, lower costs, and better long-range benefits from Ada.

CONCLUSIONS

Like any new technology, Ada imposes risks. These risks should not scare us away from using Ada; they can be managed and controlled. One of the best ways to do this is with up-front benchmarking -- trying out planned approaches before adopting them. Benchmarking will save money over the project life cycle, but it will take management and customer commitment to make it happen. If we approach projects this way, we can show that Ada does not jeopardize our chance of project success, and that it brings us the long-range benefits it promises.

Christine L. Braun is employed at SofTech, Inc., where she is Manager of Military Applications. Much of her work focuses on the effective use of Ada in DoD systems. She has been in the software engineering field for 16 years, and has been involved with Ada for 11 years. She holds a BA from Brown University and an MS from University of Toronto.

Christine L. Braun
SofTech, Inc.
460 Totten Pond Rd.
Waltham, MA 02254

Implementation of Ada Tasks on General Purpose OS's

Tasuku Miyazaki, Hirofumi Hotta, Ryuichi Yasuhara

Software Engineering Laboratory
NTT Software Laboratories

Abstract

All Ada compilers which have been developed for general purpose OS's assimilate Ada tasks into one OS task. However this implementation has some problems in I/O processing and task switching.

This paper presents a method which can implement each Ada task in one OS task to solve the above problems. One OS task (State Management task) is introduced to manage all Ada tasks to eliminate the semantics differences between Ada tasking and existing OS tasking. Some optimization methods to decrease Ada tasking overhead is also proposed.

Performance of the implementation method for real-time system models based on actual systems, such as banking systems, which are implemented by using OS tasks, are evaluated.

The following evaluation results:

(1) The execution speed of task communication using Ada without optimization is three to five times slower than those using assembly language.

(2) Using the proposed optimization method, it is possible to implement task communication using Ada which has the same approximate efficiency as those using assembly language.

1. Background

NTT has application systems such as banking systems and air traffic control systems on general purpose OS's. These are multi-task real-time systems in which many tasks are used to process entire jobs. These systems must be implemented on general purpose OS's in order to utilize a database management system of general purpose OS's. These systems also have the following two advantages because of implementation on general purpose OS's:

(1) Time-slicing facilities which are necessary for banking systems can be used;

(2) File management facilities of OS's can be used.

These systems were implemented using many task facilities. Therefore, reliability and writability can be improved by using Ada task facilities instead of using OS tasking facilities directly.

Current implementations of Ada tasks on general purpose OS's, however, generally use the implementation methods on bare machines. They assimilate all Ada tasks into one OS task. Therefore an Ada program which includes Ada tasks requires a task dispatcher for itself, and Ada tasking is processed using run-time support routines. While these implementations are relatively easy, they have the following problems:

(1) Ada task dispather overhead;

(2) asynchronous I/O;

(3) shortened time-slicing.

The main reasons for these problems is that current implementation doesn't use OS facilities efficiently.

This paper, in order to solve the above problems, proposes and evaluates the implementation method of using general purpose OS facilities without modifying of OS functions; and proposes optimizations of implementation in order to decrease Ada tasking overhead.

2. Problems of Ada task implementation on general purpose OS's

Current implementations of Ada tasks on general purpose OS's assimilate all Ada tasks in one OS task(Figure 1). This implementation is relatively easy because it does not have to differentiate between Ada tasking function and OS's tasking function; i.e., the entire tasking function is implemented by run-time support routines.

However, they have the following problems:

(1) There are dispatching overheads of Ada tasks in addition to dispatching overheads of OS tasks.

(2) In systems without asynchronous I/O, when one Ada task issues an I/O request, the whole OS task will be suspended because all Ada tasks, including executable ones, must be blocked.

(3) The time-slicing quota allocated for an OS task for Ada tasks are the same as those of other languages when tasks performing programs written by other languages are executed concurrently with Ada program tasks. In this case, the time-slicing quota for each Ada task becomes shorter than that for other OS tasks.

Therefore, the Ada tasking facility implemented by the above method can not be used for actual real-time systems.

These problems occurred because current Ada tasking implementation is only an assimilation of tasking and doesn't efficiently use OS facilities, which will be described in the following chapter.

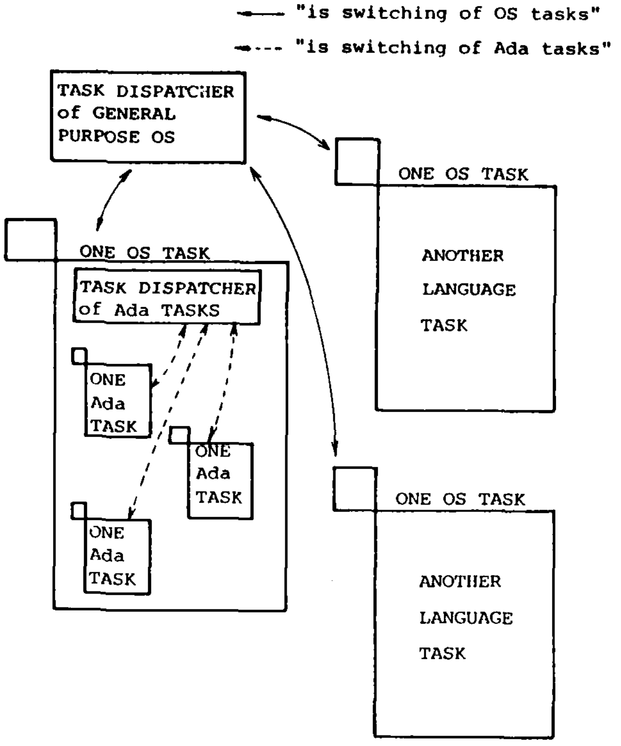

"is switching of OS tasks"
"is switching of Ada tasks"
TASK DISPATCHER of GENERAL PURPOSE OS
ONE OS TASK
ANOTHER LANGUAGE TASK
ONE OS TASK
TASK DISPATCHER of Ada TASKS
ONE Ada TASK
ONE Ada TASK
ONE Ada TASK
ONE OS TASK
ANOTHER LANGUAGE TASK

Figure 1. Original implementation of Ada tasks on general purpose OS's

3. Implementation of Ada tasks

3.1 Facilities of general purpose OS's

This paper assumes that the following functions are provided by general purpose OS's:

(1) APPEND macro (task creation)
Creates OS task. Some OS's cannot create arbitrary hierarchies of tasks.

(2) POST macro (notification of event)
Notifies the specified task of the occurrence of events. The waiting state task can be activated to change its state to ready using this macro.

(3) WAIT macro (waiting for notification of event)
Waits for the occurrence of events. The task may be changed from waiting state to ready state by using this macro.

(4) GIVE macro (request to process)
Sends a request to the queue of specified task to process.

(5) TAKE macro (take request to process)
Take data from its own task queue. This macro performs task communication using the GIVE macro.

(6) SRTW macro (set timer and wait)
After specified time passes, timer interruption will occur; this macro is used to wait for the specified time.

(7) CRT macro (cancel SRTW macro)
Cancels issued SRTW macro.

(8) PCHNG macro (change priority)
Changes priority of specified task.

(9) Inter-task common memory area
OS has a common memory area which can be used by all OS tasks.

3.2 Basic policy to solve problems

Problems described in chapter 2 can be solved by implementing one Ada task as one OS task and leaving the OS task dispatcher to dispatch Ada tasks. But the following problems stem from the differences between OS tasking facilities and Ada tasking semantics.

(1) State transition : A task which has the highest priority shall be scheduled after every rendezvous and task creation because of the semantics of Ada tasks. In order for the OS task dispatcher to schedule Ada tasks, the running task breaks execution and give the execution to the OS task dispatcher. That is to say, the state of Ada task shall be changed as executing-> ready -> executing. However, some OS's lack such state transition function.

(2) Task creation : In Ada, arbitrary task hierarchies can be created, that is, any Ada task can create a Child Task.
It is necessary for OS's to be able to create arbitrary hierarchies of tasks, if the OS task creation facility is used to create Ada tasks; however, some OS's lack such a facility.

(3) File sharing : Some OS's have the limitation that a task which opened a file must read, write and close that file. Ada tasks lack such a limitation. Therefore, if more than two Ada tasks want to access the same file, one OS task must access files to that file instead of each task.

To solve the above problems, a management routine which manages Ada task dispatching, task creation and file sharing must be introduced for one Ada program.

There are other problems in implementing Ada tasks besides the above, such as the processing of shared variable and exception handling.

3.3 Implementation of state management routine

We can take the three implementations of state management routines, i.e. (1)task, (2)SVC and (3)subroutine. But it is difficult for SVC routines and subroutines to dispatch Ada tasks and share files. Therefore task management routines are implemented as an OS task, called the "State Management Task". Ada tasks send a request to the State Management Task to process any facilities which the original OS's lack.

3.4 Implementation of Ada tasks

In this chapter, an implementation method for Ada task facilities, mainly task creation and task communication (rendezvous), which depend on each OS's facilities and influence the performance of system, are discussed.

3.4.1 Task creation

All Ada tasks are created by the State Management Task not by the Ada task directly. The State Management Task issues a task creation macro and creates a new OS task for the Ada task. All information which includes relation between Parent and Child Tasks, Siblings of Ada tasks and other information necessary for managing Ada tasking is stored in the user area of the task control block(TCB)(Figure 2).

The steps to create Ada tasks are as follows:

(1) The Ada task which wants to create a Child Task (Parent Task) sends a request to the State Management Task for task creation using the POST macro.

(2) The Parent Task issues a WAIT macro to change its own state from executing to waiting.

(3) The State Management Task issues an APPEND macro and creates a task (Child Task).

(4) The State Management Task sets the task name in the TCB of a Parent Task.

(5) The State Management Task sets the parent-child and sibling relation information in the user area of Child Task's TCB.

(6) If the request is to create more than two tasks, each Child Task issues a WAIT macro to wait until the creation of all Child Tasks.

(7) After all Child Tasks are created, the State Management Task issues a POST macro to all Child Tasks to activate them.

3.4.2 Rendezvous

The next steps outlines the implementation of a rendezvous(Figure 3).

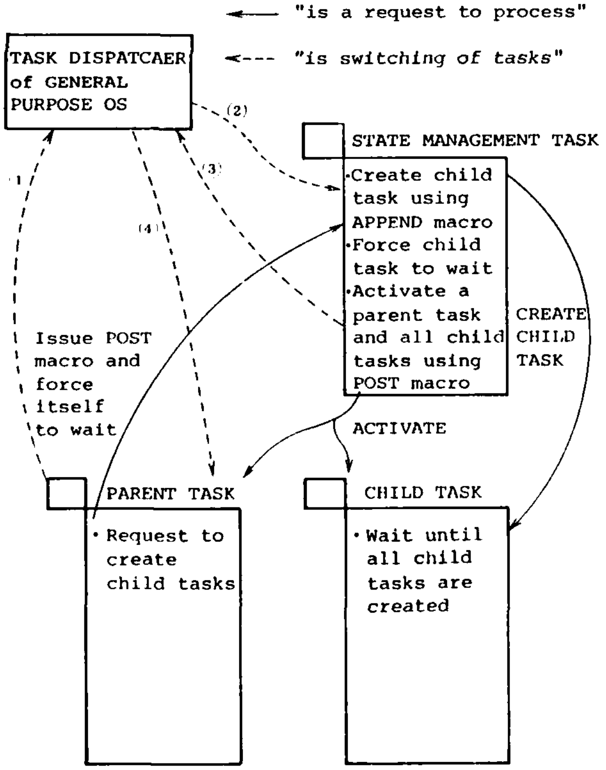
(1) The semantics of an Ada rendezvous require implementation to be a synchronous communication. Calling Task(entry call side) issues GIVE and POST macro to send data and Called Task(accept side) issues WAIT and TAKE macro to receive data.

(2) When Called Task is ahead of a rendezvous request from Calling Task, Calling Task changes the priority of Called Task, if necessary, and issues a WAIT macro. Then, Called Task takes, the data and begins the rendezvous.

When Calling Task is waiting for Called Task to become ready, Calling Task issues only the WAIT macro. Eventually when Called Task is ready, it changes the priority of itself, if necessary. Then it takes the data and begins the rendezvous.

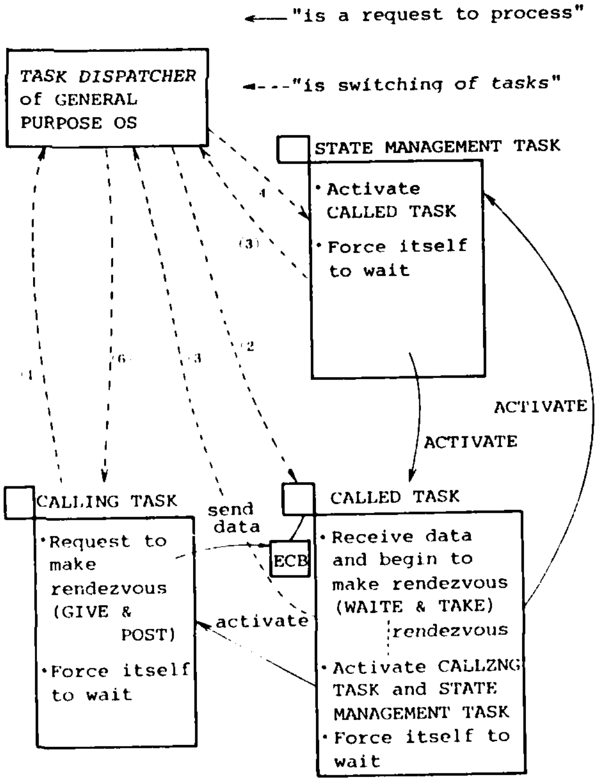
(3) When the rendezvous ends, Called Task issues a POST macro to the Calling Task to change Calling Task from the waiting state to the ready state.

(4) If the priority of Calling Task is higher than a Called Task's, The Called Task can not continue execution because of Ada semantics(see Ada reference manual 9.8). Therefore at the end of rendezvous, Called



← "is a request to process"
←--- "is switching of tasks"
TASK DISPATCAER of GENERAL PURPOSE OS
(1)
(2)
(3)
(4)
STATE MANAGEMENT TASK
·Create child task using APPEND macro
·Force child task to wait
·Activate a parent task and all child tasks using POST macro
CREATE CHILD TASK
Issue POST macro and force itself to wait
ACTIVATE
PARENT TASK
· Request to create child tasks
CHILD TASK
· Wait until all child tasks are created

Figure 2. Implementation of creating Ada tasks



← "is a request to process"
←---"is switching of tasks"
TASK DISPATCHER of GENERAL PURPOSE OS
STATE MANAGEMENT TASK
· Activate CALLED TASK
· Force itself to wait
(1)
(2)
(3)
(4)
(6)
ACTIVATE
ACTIVATE
CALLING TASK
· Request to make rendezvous (GIVE & POST)
· Force itself to wait
send data
ECB
activate
CALLED TASK
· Receive data and begin to make rendezvous (WAITE & TAKE)
rendezvous
· Activate CALLZNG TASK and STATE MANAGEMENT TASK
· Force itself to wait

Figure 3. Implementation of rendezvous

Task must issue a WAIT macro to break execution.

However there is no task for activating the waiting Called Task, therefore the State Management Task is introduced and it issues a POST macro to the Called task. The Called Task restores its task priority and issues a POST macro to the State Management Task so that State Management Task can issue a POST macro to the Called Task later. Then Called Task issues a WAIT macro to force itself to wait.

(5) The activated State Management Task issues a POST macro to the Called Task. Then it issues a WAIT macro to force itself to wait. At this time, the OS task dispatcher switches to the task having the highest priority. Therefore the task having the highest priority will be executed first. The State Management Task shall be given a higher priority than any other Ada task.

3.4.3 Access to shared files

Not all Ada tasks issue I/O macros directly. Each Ada task sends a request to the State Management Task to open, close, read or write. The State Management Task receives the I/O request and issues an I/O macro in behalf of Ada tasks.

3.4.4 Access to shared variables

Ada reference manual specifies that "if two tasks read or update a shared variable, then neither of them may assume anything about the order in which the other performs its operations except at the point where they *synchronize*"[1]. Tasks can synchronize only at the start and at the end of their rendezvous, at the start and at the end of its activation, and at the completion of their execution.

Furthermore, a program that uses shared variables is erroneous if the following are violated:

"(a) If between two of its synchronization points a task reads a shared variable, then the variable is not updated by any other task at any time between these two points.

(b) If between two of its synchronization points a task updates a shared variable, then the variable is neither read nor updated by any other task at any time between these two points."[1].

Shared variables were implemented as follows to *satisfy conditions* (a) and (b):

(1) The variables of a task which creates Child Tasks is placed in a common area among tasks.

(2) When the Child Task is created, the Parent Tasks' variables which are used by the Child Task are copied into Child Task's own area.

(3) When the shared variables are updated, only the variable copy is updated and its copy is copied back into the common area at the next synchronization point. The variables which are not updated are copied from the common area at each synchronization point.

3.4.5 Exception handling

When an exception is raised in a task, it is not usually propagated to other tasks. But an exception can be propagated to a Calling Task if it is raised during a *rendezvous*.

Implementation of exception handling is as follows:

The interruption handler of each task includes a routine to terminate rendezvous. If the exception occurs during rendezvous, the interruption handler sets a flag which indicates that an exception has been occurred and performs the rendezvous termination processing, i.e. changing priority or notification of rendezvous end to the Calling Task. This flag is placed in the task common area and exception existence during rendezvous is checked both after entry call statement and accept statement execution. If this flag indicates that an exception was raised, an exception handler corresponding to the exception is executed.

4. Evaluation

4.1 Model for evaluation

Real time processing, one type of concurrent processing is classified into two types[2][3]. The first is a transaction-type system which is used for on-line systems such as banking systems. The second is a control type which is used for air traffic control systems.

4.1.1 The transaction type model

The transaction-type system deals with data which is input at irregular intervals. To deal with irregular intervals' input data, these data are queued in a buffer and each task performs the processing according to the data taken from buffer. This implementation is used for banking system, and has the following characteristics:

(1) A queue to deal with data which is input at irregular intervals.

(2) Task construction and task communication of these system types are almost identical. Therefore, a typical model can be easily defined.

(3) It is not necessary to determine the upper limit of response time strictly but, of course it is better to have a better response time.

When this system is described using Ada, it is necessary to implement an asynchronous communication using a queue. Therefore, as is well known, a buffer task to implement asynchronous communication must be introduced.

In this paper, the communication processing part of banking systems was examined and modeled. The banking system is used for two types of tasks. The first type accepts input data and delivers it to the second type of task, which processes these data according to its contents. Usually banking systems consist of one first-type task and several second-type tasks. These two types of tasks communicate with each other. Processing of a transaction-type model using an assembly language is illustrated in Figure 4. An illustration of the same system using Ada, instead of the assembly language, is shown in Figure 5.

4.1.2 Control-type model

Control-type model systems deal with data which is input at regular intervals. To deal with regular interval input data, each task must complete its processing within this interval. Consequently, it is not necessary to use a queue in the control-type system, unlike the transaction system. This implementation is used for air traffic control systems by NTT, and has the following characteristics:

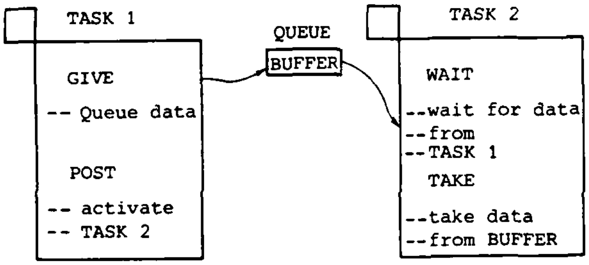
(1) The upper limit of a response time is strictly determined based on input time intervals.

(2) Task construction and task communications differ from system to system.

(3) It is not necessary to use the queue for the task communication.

When this system uses Ada, buffer task is unnecessary and task communication can be implemented with the rendezvous.

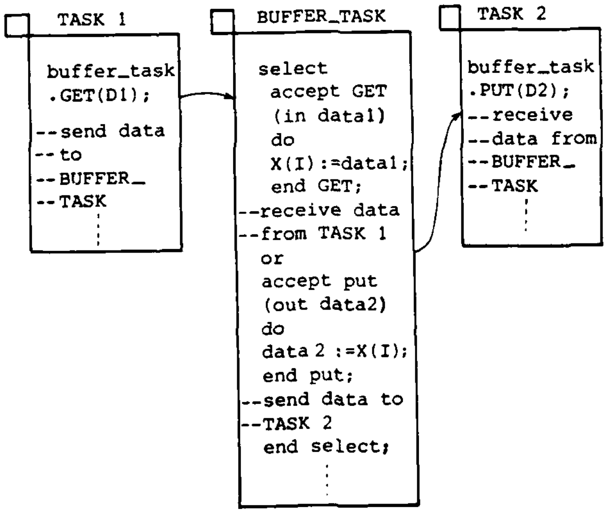
The transaction system is modeled using a fixed form of task construction. On the other hand, there are various types and relations of task construction in the control-type systems. Hence, it is difficult to determine one typical model. Therefore, two sub-models are used for evaluation. One sub-model has a task which sends the data to two tasks. Another sub-model has tasks which send data to a single task. The first sub-model using assembly language is shown in Figure 6. The same sub-model using Ada is shown in Figure 7.



```
GIVE
-- Queue data

POST
-- activate
-- TASK 2
```

```
WAIT
--wait for data
--from
--TASK 1
TAKE
--take data
--from BUFFER
```

Figure 4. Implementation of transaction-type model using assembler



```
buffer_task
.GET(D1);
--send data
--to
--BUFFER_
--TASK
   :
```

```
select
  accept GET
   (in data1)
   do
   X(I):=data1;
   end GET;
--receive data
--from TASK 1
 or
 accept put
 (out data2)
 do
 data2 :=X(I);
 end put;
--send data to
--TASK 2
 end select;
     :
```

```
buffer_task
.PUT(D2);
--receive
--data from
--BUFFER_
--TASK
   :
```

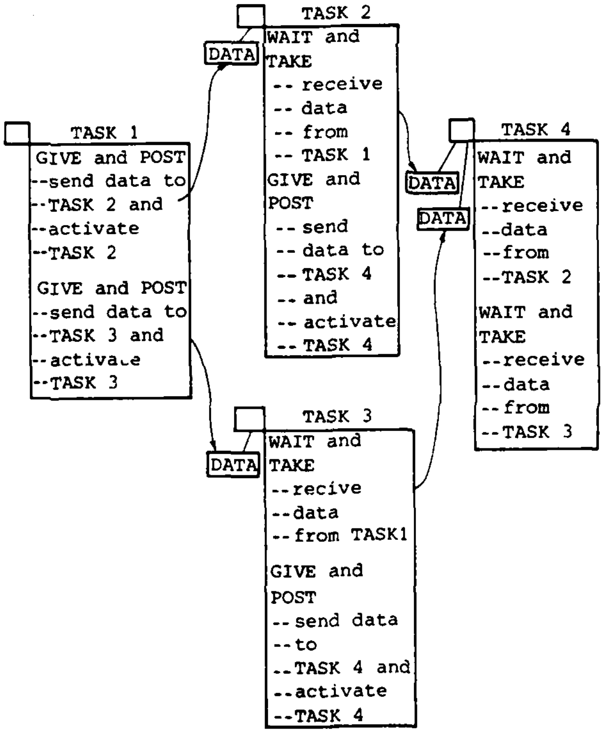
Figure 5. Implementation of transaction-type model using Ada tasks

4.2 Evaluation method

To make a quantitative performance evaluation, the number of executed machine instructions in task management and task switching processing are measured for two cases; one where the above implementation method is used, and another where OS macros are directly used using assembly language. The number of machine instructions is assumed to be the same for parts other than for task communication processing in both cases.

The number of each OS macros executed machine instructions are known. In addition, execution times of each macro have been estimated for each model. Thus the total number of executed machine instructions for each model can be calculated.

The results of the performance evaluation are shown in Table 1. The "Synchronous Ada model" column in the transaction-type model shows the model in which task construction is the same as the original, however, task communications are performed synchronously. This model is presented only for the evaluation of buffer task overhead.



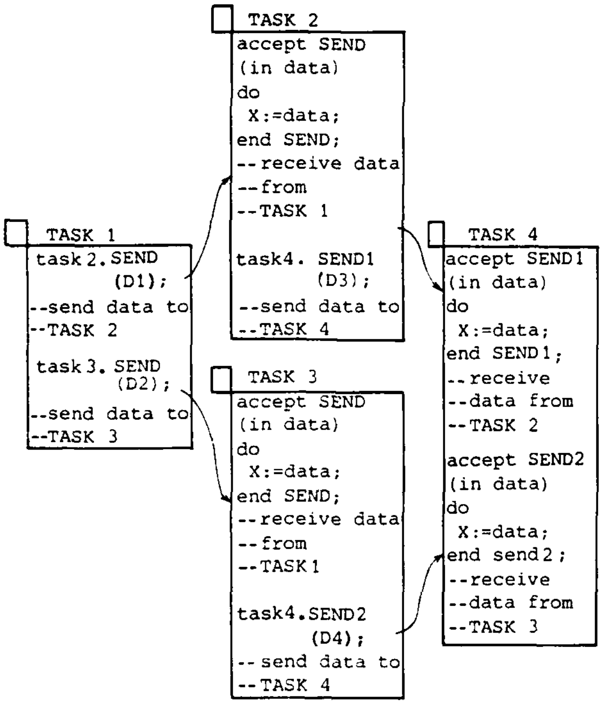
```
GIVE and POST
--send data to
--TASK 2 and
--activate
--TASK 2
GIVE and POST
--send data to
--TASK 3 and
--activate
--TASK 3
```

```
WAIT and
TAKE
 -- receive
 -- data
 -- from
 -- TASK 1
GIVE and
POST
 -- send
 -- data to
 -- TASK 4
 -- and
 -- activate
 -- TASK 4
```

```
WAIT and
TAKE
--recive
--data
--from TASK1
GIVE and
POST
--send data
--to
--TASK 4 and
--activate
--TASK 4
```

```
WAIT and
TAKE
 --receive
 --data
 --from
 --TASK 2
WAIT and
TAKE
 --receive
 --data
 --from
 --TASK 3
```

Figure 6. Implementation of control-type model 1 using accembler



```
TASK 1
task2.SEND
      (D1);
--send data to
--TASK 2

task3.SEND
      (D2);
--send data to
--TASK 3
```

```
TASK 2
accept SEND
(in data)
do
 X:=data;
end SEND;
--receive data
--from
--TASK 1

task4. SEND1
        (D3);
--send data to
--TASK 4
```

```
TASK 3
accept SEND
(in data)
do
 X:=data;
end SEND;
--receive data
--from
--TASK1

task4.SEND2
        (D4);
--send data to
--TASK 4
```

```
TASK 4
accept SEND1
(in data)
do
 X:=data;
end SEND1;
--receive
--data from
--TASK 2

accept SEND2
(in data)
do
 X:=data;
end send2;
--receive
--data from
--TASK 3
```

Figure 7. Implementation of control-type model 1 using Ada tasks

4.3 Evaluation results

The following conclusions result from evaluation of Table 1.

(1) Comparison of the two control-type sub-models revealed that there are 3 times number of executed machine instructions using Ada as compared to the assembly language. There is no overhead differences between the two models.

(2) There are 5 times the number of instruction steps using Ada as when using the assembly language for the transaction model because of the extra buffer task needed to implement asynchronous communication. The overhead using this buffer task accounts for about 50 percent of all overhead.

(3) The number of task switchings are significant overheads for both type models.

5. Optimization

As is seen in the evaluation results, Ada tasking overhead is rather large. A rendezvous optimization technique for the implementation of Ada tasking is proposed.

There are overheads using a buffer task on transaction type models using Ada. Buffer task overhead is optimized by using the Habermann and Nassi method[4][5]. In their optimization, all buffer task processing is changed into subroutines. This optimization can be applied to the transaction type system, however, it can not be applied to the control-type system because of the following reason. If the Called Task includes I/O requests or delay statements, the Calling Task must wait for the completion of these I/O requests or delay statements. Waiting for these I/O requests or delay statements becomes overhead for Calling Task. In the control systems, entire jobs are broken into several tasks so as that they can be executed concurrently. Therefore, many tasks have I/O requests or delay statements. The optimization method applicable to systems in which tasks have I/O requests or delay statement like control-type system are described here.

Table 1. Number of executed machine instructions

TYPE of MODELS / FUNCTIONS	TRANSACTION TYPE MODEL			CONTROL TYPE MODEL			
	MODEL 1		SYNCHRONOUS Ada MODEL	MODEL 1		MODEL 2	
	ASSEMBLER	Ada		ASSEMBLER	Ada	ASSEMBLER	Ada
TASK DISPATCH	103×2 =206	103×9 =927	103×5 =515	103×4 =412	103×17 =1751	103×4 =412	103×13 =1339
ENTRY CALL	94×1 =94	168×2 =336	168×1 =168	94×4 =376	168×4 =672	94×3 =282	168×3 =504
BEGNNING OF ACCEPT	110×1 =110	110×2 =220	110×1 =110	110×4 =440	110×4 =440	110×3 =330	110×3 =330
END OF ACCEPT	—	186×2 =372	186×1 =186	—	186×4 =744	—	186×3 =558
STATE MANAGEMENT TASK	—	130×2 =260	130×1 =130	—	130×4 =520	—	130×3 =390
TOTAL	410	2115 (5.2)	1239 (3.0)	1228	4127 (3.4)	1024	3121 (3.0)

* The valves in parentheses are the ratio of Ada to assembler

In the control-type systems, Calling Task receives the data from another task or hardware device and sends them to Called Task. Called Task receives data from Calling Task, processes it and write the results in the data bases. In this model, Calling Task does not a wait the response from Called Task after sending data. Consequently, data communication between the two tasks is only transmitting data and it is not necessary bidirectional data communication such as rendezvous. Therefore, when nothing but sending the data is used by rendezvous, the Calling Task does not have to wait the completion of rendezvous. That is, in this case, rendezvous can be implemented as an asynchronous communication and a lot of task switching can be eliminated.

The described optimization can be used under the following conditions:

(a) The accept statement has only in mode parameters.

(b) Execution of the accept statement can be done by the Calling Task. That is to say, the variables used in accept statements can be used by the Calling Task or if the accept statement has no sequences of statements with it.

(c) The priority of Calling Task is higher than or equal to that of Called Task.

(d) Accept statements do not include I/O requests or delay statements.

If the system satisfies these conditions, Calling Task can execute the process of accept statements in its context of Calling Task and send the results of this processing to Called Task. Therefore, rendezvous can be implemented without task switching. Almost all control systems satisfy the above conditions.

The details of implementing the described optimization are as follows(Figure 8):

(1) When Calling Task is going to execute an entry call and Called Task is not waiting at the point of rendezvous, the rendezvous processing described in chapter 3 is executed as usual.

(2) When Calling Task is going to execute an entry call and Called Task is waiting for rendezvous, the former executes the accept statements as a subroutine.

(3) When execution of accept statements is finished, Calling Task sends the data processed in the subroutine of accept statements to Called Task using GIVE and POST macro. Calling Task continues to execute without task switching.

(4) Called Task take the data from ECB and continues to execute.

In control-type systems, Called Tasks are usually waiting for data from Calling Tasks. Therefore, unnecessary task switching can be avoided by using the described optimization, and asynchronous communication can be implemented using Ada. After optimization, performance of this communication is the same as when an assembly language is used.

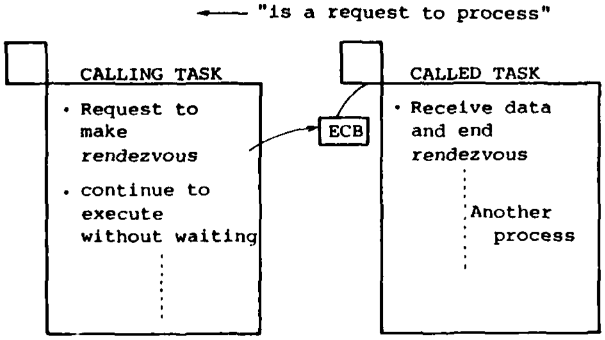
6. Conclusion

In this paper, the implementations of Ada tasks on general purpose OS's was discussed. The following conclusions were reached:

(1) Ada task implementation was designed so that Ada tasks were created by OS's task creation macro and task communication was performed by OS task communication macros. It was found that usual general purpose OS lacks enough facilities to implement Ada tasks(the facility to manage the parent-child relation of Ada tasks and file sharing.) and that a State Management Task is required. By this method, Ada tasks can be implemented as an OS task without modification of the OS.

(2) In this implementation, there was considerable overhead in switching task at the end of rendezvous. Consequently, it has overhead 3.0 to 5.0 times greater than an assembly language.

(3) An optimization method was proposed to decrease the execution time of the task communication of control-type systems. Using the optimization, Ada tasks were implemented which had the same performance as the assembly language.



"is a request to process"
CALLING TASK
· Request to make rendezvous
· continue to execute without waiting
ECB
CALLED TASK
· Receive data and end rendezvous
Another process

Figure 8. Implementation of rendezvous (optimized version 1)

In this paper, the number of machine instructions was used to evaluate performance. A future application is to implement Ada tasks on general purpose OS's based on the proposed implementation and to measure actual execution time in order to examine its efficiency.

7. Acknowledgement

The authors wish to acknowledge the assistance and advice of Ryoichi Hosoya, Dr. Tsuneo Furuyama and Shigeru Nishiyama. Appreciation is also due to Dr. Shuetsu Hanata for his encouragement and helpful suggestions.

References

[1]Department of Defense, ANSI/MIL-STD-1815A "Reference Manual for the Ada Programming Language" 1983

[2]T.Mitsumaki and H.Kuwahara "Real-time Processing Technologies in Process Computers"(in Japanese) CORONA 1986

[3]Y.Ono "Software of On-line System"(in Japanese) Sangyou Tosyo 1977

[4]A.Burns "Concurrent Programming in Ada" Cambridge University Press 1985

[5]A.N.Habermann and I.R.Nassi "Efficient implementation of Ada tasks" Tech Rep CMU-CS-80-103, Carnegie-Mellon Univ 1980

The Authors

Tasuku Miyazaki
Software Engineering Laboratory, NTT Software Laboratories Nippon Telegraph and Telephone Corporation, NTT Shinagawa TWINS 1-9-1, Kohnan Minatoku, Tokyo 108 JAPAN

Research Engineer in the Software Engineering Laboratory. Mr. Miyazaki received his B.S. and M.S. from the University of Iwate in 1983 and 1985. His research interests include programming language design.

Hirofumi Hotta
Software Engineering Laboratory, NTT Software Laboratories Nippon Telegraph and Telephone Corporation, NTT Shinagawa TWINS 1-9-1, Kohnan Minatoku, Tokyo 108 JAPAN

Senior Research Engineer in the Software Engineering Laboratory. Mr. Hotta received his B.S. and M.S. from the University of Osaka in 1978 and 1980. He is currently engage in research of Programming Language.

Ryuichi Yasuhara
Software Engineering Laboratory, NTT Software Laboratories, Nippon Telegraph and Telephone Corporation, NTT Shinagawa TWINS 1-9-1, Kohnan Minatoku, Tokyo 108 JAPAN

Senior Research Engineer, Supervisor in the Software Engineering Laboratory. Mr. Yasuhara received his B.S. and M.S. from the University of Kyoto in 1972 and 1974. His research work and interests are in the design of system Programming Language and Programming environment.

OPPORTUNITIES FOR PARALLELISM IN ADA: LANGUAGE SUPPORT AND RESTRICTIONS ON EXPLICIT AND AUTOMATIC PARALLELISM

Carol R. Peo

Alliant Computer Systems Corporation

Modern architectures support parallelism at the instruction, expression, loop, block and task level. This paper discusses features required for explicit parallelism and techniques used for automatically detecting parallelism at each of these levels. Ada is examined in terms of how it promotes and inhibits parallelism. Ada's problems as well as its unique advantages for parallelism are presented. The paper shows that Ada can exploit many forms of parallelism.

1. INTRODUCTION

Parallel processing yields tremendous performance gains over serial processing. Machine architectures have become increasingly sophisticated in their hardware support for parallel processing. Hardware now supports parallelism in many forms: multi-processing with multiple processors or functional units (termed MES or multiple execution of scalar code), vector processing (termed SEA or single execution of array operations), and multi-processing vector processing (termed MEA or multiple execution of array operations). Machines based on these forms of parallelism offer enormous potential speedups over traditional Von Neumann architectures.

Languages and language processors should take advantage of the opportunities afforded by these parallel architectures. Two distinct approaches towards program parallelism have emerged in recent years. One approach has aimed at explicit high-level parallelism, directed towards relatively large independent sections of code. The other approach has focussed on more fine-grained, implicit parallelism. Languages which provide concurrent programming constructs such as Ada and Modula-2, exemplify the first approach, while FORTRAN compilers which automatically generate vector and parallel code exemplify the latter. Languages should take advantage of all levels of parallelism to maximize performance gains. This paper will discuss the various forms of parallelism, how they can be exploited, and particularly, how they relate to Ada.

Section 2 will discuss levels of parallelism. Section 3 will discuss explicit parallelism. Section 4 will present a brief survey of techniques used for automatic detection of implicit parallelism and will discuss issues that make such detection problematical. Section 5 will examine Ada in light of what it provides and lacks to support parallelism. Section 6 will draw some conclusions.

2. LEVELS OF PARALLELISM

Parallelism can be applied at each of the following levels within a program:

instruction level
expression level
loop level
block level
task level

Instruction Level - Pipelined architectures which overlap the decoding, fetching and execution stages of instruction processing have been available for a long time. Compilers can schedule instructions to minimize the pipeline stalls that occur when required data is not available.

Expression Level - MES and MEA architectures can take advantage of expression-level parallelism. This refers to the parallel evaluation of operands on the right-hand-side (RHS) of an expression. Each independent expression can be executed in parallel. Transformation techniques using associativity, commutativity and distributivity can be applied to increase the amount of parallelism. Techniques promoting tree-height reduction enhance the parallelism at this level [14,15]. This fine-grained parallelism by itself may not provide significant performance improvement, since it can easily be impeded by a memory system. Memory clashes can overwhelm the effect of parallelism at this level.

Loop-Level - SEA and MEA architectures can exploit loop-level parallelism. Iterations of a loop are executed in parallel by vector hardware or by doling out iterations of the loop to multiple processors, or a combination of the two. The loop

```
for index in index'range loop
    A(i) := B(i) + C(i);
end loop;
```

can be processed by a vector processor in a single instruction, which might be represented as:

```
A(index'range) := B(index'range) +
    C(index'range);
```

Alternatively, the loop can be processed in parallel by multi-processors, where one processor executes iteration 1, another processor executes iteration 2, and so forth. This style of self-scheduled loop execution has been termed concurrency. Some loops cannot be vectorized because of data dependencies, but can be run in parallel using concurrency (data dependencies will be discussed in Section 4). Additionally, some nested loops can utilize both vectorization and concurrency by using vector mode on the inner loop and concurrency on the outer loop. For example:

```
loop1:
for i in one'range loop
    loop2:
    for j in two'range loop
      A(i,j) := B(i,j) + C(i,j);
    end loop2;
end loop1;
```

In this example, iterations of loop 1 could be distributed to different processors, and within each processor, vector operations could execute loop 2. If one'range were 1..8 and two'range were 1..32, for example, this loop when run in concurrent-outer-vector-inner (COVI) mode on a machine with eight processors, would require just one add instruction on each of the eight processors as opposed to 512 add instructions that would be required for this loop to be run serially.

Concurrency is also used for loops that cannot be vectorized. For example, the range of a WHILE loop construct is not as well-defined as that of a FOR loop. WHILE loops often do not use an induction variable, and frequently use EXIT statements to exit from the loop when a condition is reached. Such loops are difficult to vectorize since processing must be synchronized at the conditional statement to know whether the loop should continue. Concurrency has access to synchronization mechanisms and can execute such a loop in parallel.

```
loop1:
WHILE  true loop
     IF condition then exit loop1;
     statements
end loop1;
```

This loop could execute in concurrency mode assuming the statements are capable of being run in parallel, with a synchronization point set at the IF statement.

Other constructs which prohibit vectorization but not concurrency include multi-way branching inside a loop, forward branches outside the loop, and return statements.

Hardware support is required for concurrency. In particular, the hardware should efficiently support data synchronization, communication and processor self-scheduling. The Alliant FX/8, for example, has hardware registers to support processor self-scheduling, synchronization registers and a high-speed concurrency bus.

Block Level - MES and MEA architectures can also take advantage of block level parallelism. This form is applied to adjacent statements or loops. A block of adjacent assignment statements, the evaluation of the RHS's of a block of assignment statements with intervening conditional statements, statements adjacent to loops, and consecutive loops could each be executed using concurrency on multiple processors if there were no data dependencies or if synchronization points could make the data dependencies manageable.

Task Level - Another form of parallelism available to MES and MEA architectures is task level parallelism. Task level parallelism refers to the concurrent execution of disjointed portions of code. This may refer either to subprograms which can be executed in parallel if there are no data dependencies, or to explicitly parallel code such as Ada tasks. This level of parallelism is achieved by executing "tasks" on separate processors.

All of these levels of parallelism can be applied in concert to a single program. When conflicts for resources arise, the resource allocation choice might be made based on which mode provides the greater performance speedup. Alternatively, under user control, the choice could be biased towards satisfying explicit parallel constructs first, and allocating only the remaining resources to implicit parallelism. Thus on a machine which supports all of these forms of parallelism, portions of programs might run in any of the following modes:

scalar (serial)
vector
vector-concurrent
scalar-concurrent
multi-processing/scalar
multi-processing/vector
multi-processing/vector-concurrent
multi-processing/scalar-concurrent

The Alliant FX/8 is an example of an architecture that supports all of these forms. An Ada program could be executed such that each task executed on its own processor, and the code within each task executed in parallel using both vectors and concurrency. Concurrency could be utilized if processors were available because either there were fewer tasks than processors or because some of the tasks were blocked thus making the processors available for concurrency.

3. EXPLICIT PARALLELISM

Parallelism can be either explicit or automatic. That is, either the programmer controls precisely which parts of the program are run in parallel, or the programmer allows the compiler to automatically detect where parallelism can be achieved. Additionally we can have hybrids of these two, where tools are provided which help the programmer structure the code such that parallelism is enhanced. The programmer supplies information to the tool, which inserts pragmas in the code instructing the compiler to generate paralle code. Additionally, the tool helps the programmer restructure the code in such a way as to enhance the opportunities for parallelism. Tools to help FORTRAN programmers in this way are already available.

Explicit parallelism has the advantage that the programmer controls where parallelism is applied, and languages can define precise semantics for it. This section will outline language constructs that promote explicit parallelism. These language constructs include task-level constructs such as Ada's tasking, vector constructs like those added to FORTRAN/8X and FX/FORTRAN, and pragmas that can make any level of parallelism explicit.

TASKS

Tasks identify sections of code which can execute concurrently. Ada tasking provides an ideal vehicle for high-level parallelism because it explicitly informs the compiler and runtime system what code can execute in parallel, and precisely defines the semantics for the behavior of multiple tasks in a program. Other languages have provided means for subprograms to run independently. Some implementations of FORTRAN, for example, support macrotasking for this purpose.

In addition to performance gains achievable through task-level parallelism, the use of tasks has advantages from a design standpoint. Many applications comprise a set of concurrent and relatively independent functions. With tasking, these applications can be designed as single programs with multiple tasks, thus extending the protection of visibility and typing rules to the entire application. Finally, the use of tasks rahter than system-dependent services and processes promotes portability and inter-operability.

VECTOR CONSTRUCTS

Specific language constructs for loop-level parallelism provide the programmer a means other than loops to specify operations to be applied to multiple elements of a vector. Such constructs instruct the compiler explicitly to generate parallel code for the operations. The semantics of a vector operation versus a loop executed in serial mode specify that all the operands are fetched once before the vector operation rather than on each iteration of a serially-executed loop. Using explicit language constructs to specify such behavior, makes the design clearer and asserts that such behavior is correct.

FORTRAN/8X and FX/FORTRAN include many vector extensions to support loop vectorization. Constructs of importance include:

Vector types

The language recognizes vectors as a type. Assignment is defined for vectors, and objects of vector type can be passed as parameters to procedures and be returned from functions. Vectors are used for each dimension in multiply-dimensioned array.

Vector expressions

Several ways to reference vectors are useful for explicit loop-level parallelism. FORTRAN/8X allows users to reference vectors by range, length and wild carding, all with an optional stride. Stride allows array sections to be named. Wild carding is required to select sections and shifted sections of arrays, such as all columns, all rows, or the area above or below the diagonal. The following FORTRAN expressions are vector references for A:

A(1:32:1)	--	Elements 1..32
A(1:32:2)	--	Odd elements 1..31
A(3:#10)	--	Elements 3..12
A(*)	--	All elements of first dimension
A(i, *)	--	All elements of second dimension
A(i, * + 2)	--	All elements in (lower_bound+2 .. upper_bound) of second dimension
A	--	All elements of A

FORTRAN/8X also allows array references using an index vector to select those elements referenced in the index vector.

A(V(*)) -- Elements of A corresponding to the value of the elements of V.

For example, if V = <0,3,5,6>, then this A(V(*)) references A(0), A(3), A(5), A(6). This kind of expression is useful to reference sparse matrices.

Boolean vectors can also be used to select elements in an array.

A(B(*)) -- Selects elements of A corresponding to the elements of B whose value is TRUE.

For example, if B = <True, True, False, True>, then A(B(*)) selects A(0), A(1), and A(3). This expression is useful for conditional assignments which will be described below.

Vector operations

All arithmetic operations should be predefined operations for vector and array types. Thus A(*) + B(*) would yield a result vector whose elements contained the results of the elementwise sum of vectors A and B. Another set of useful arithmetic operations take one operand of type vector, and one of scalar type. The operator is applied to each element of the vector and the scalar object. Thus A(*) + 5.0 would produce a result vector whose elements contained the results of an add of each element and 5.0.

Reduction operations are also useful for explicit loop-level parallelism. SUM and DOT-PRODUCT are examples of reduction operations.

Vector assignment and conditional assignment

FORTRAN/8x added semantics for treating arrays as aggregates for assignment. FORTRAN/8x also defines conditional assignment for arrays. Its syntax is as follows:

```
WHERE (A(*).NE.0)
  B(*) := C(*)/A(*);
OTHERWISE
  B(*) := 0;
```

The effect of this statement would be to assign the result of the C/A to B for all elements of A which are non-zero. Such a construct replaces the loop:

```
for i in index'range loop
    if A(i) /= 0 then
       B(i) := C(i) / A(i);
  end if;
end loop;
```

Below is an example of a code fragment coded using loops, and one using predefined vector operations. Clearly the second is preferrable in terms of clarity, and, on vector hardware, will perform much faster than the first.

```
for i in index'range loop
     for j in subindex'range loop
```

```
        A(i,j) := B(i,j) * C(i, j)/ D
    end loop;
end loop;
```

(for simplicity we will assume that index and subindex refer to the type used in the declaration of A.)

```
A := B * C/D;
```

PRAGMAS

Pragmas that direct a compiler to generate parallel code are another form of explicit parallelism. A pragma such as CONCURRENT(subprogram) might inform the compiler that the subprogram can be executed in parallel. This would allow loops with embedded calls to subprograms to execute concurrently. The programmer would have to ensure that the concurrent subprogram had no dangerous side-effects such as the modification of global data. Subprograms that exhibit such behavior when specified as CONCURRENT are erroneous just as are tasks which modify shared data outside synchronization points.

4. TECHNIQUES FOR AUTOMATIC PARALLELISM

Automatic vectorization and concurrency promotes portability and transparency since parallel code is generated for programs written without concern for parallelism. There has substantial research into the problems of automatic vectorization and concurrency. The theoretical foundation was laid by Kuck in [14, 15] and Allen and Kennedy [1, 3]. An excellent overview of the technology can be found in [19]. FORTRAN compilers that make use of this technology are already available. Alliant Computer Systems Corporation's FX/FORTRAN automatically generates vector and concurrent code. This section will briefly present the major techniques used to detect opportunities for parallelism.

The basic factor that determines if a section of code can be run in parallel mode is the existence of data dependencies within that section. Since a vector operation fetches all of the source operands before the operation or assignment is executed, code which modifies data that is used on later iterations of a loop cannot be vectorized. It might, however, still be a candidate for concurrency. For example, the following loop:

```
    for i in index'range loop
        A(i) := B(i) + A(i-1);
    end loop;
```

can not be vectorized since each iteration of the loop depends on a value of A that was calculated during the previous iteration. If the code were vectorized, incorrect values of A would be used in the calculation.

When executed in concurrent mode on multiple processors, synchronization points can be inserted before the reference to A(i-1) on each iteration to force the processor executing the iteration to wait until the updated data is available. This enforces a partial ordering on the loop. Additionally, compilers can sometimes schedule instructions such that the data is always available at the time it is needed.

Automatic parallel code detection focuses on data dependency analysis and program transformation techniques that eliminate data dependencies. There are three forms of data dependency.

True or flow dependence: As in the above example, one statement or iteration of a loop produces a result that is used by a later statement or iteration.

```
A := X;
Y := A;
```

Anti-dependence: One statement reassigns a value to a variable which is referenced by a previous statement. Here again, if the statements are executed in parallel, the first statement might retrieve an incorrect value. In the following example, A must be assigned the original value of X.

```
A := X;
X := Y;
```

Output dependence: Two statements assign new values to the same variable. When not executed in serial order, the variable might contain the incorrect value at the end of statements' execution. After execution of the following statements, A must contain Y, not X.

```
A := X;
A := Y;
```

Output and anti-dependence are not true dependencies since they result only because of the use of a common variable.

If different variables were used to hold the results, the statements in 2 and 3 above could be run in parallel.

```
        A := X;
        New_X := Y;
```

and

```
        A := X;
        New_A := Y;
```

Now there is no data dependency in these pairs of statements and they can be executed in parallel. Of course, all references to X and A following these substitutions would have to refer to new_X and new_A respectively.

Another kind of dependency is control dependence where conditional branches can alter the course of execution.
In the following code fragment an IF statement controls whether the assignment statement is executed.

```
    for i in index'range loop
        if condition then
            A(i) := B(i);
        end if;
    end loop;
```

A solution to this problem is to introduce a variable that contains the boolean value of the conditional expression. That variable then can guard the assignment statement. This technique converts the control dependence in this loop to data dependence, which can then be manipulated using normal data dependency techniques. The transformed loop appears as

```
    for i in index'range loop
        B := condition;
        WHERE (B) A(i) := B(i);
    and loop;
```

Many techniques have been developed to reduce data and control dependencies by transforming programs so that more code can be run in parallel. Where data dependencies still remain, code can often be executed using concurrency on multi-processors. The following section mentions a few techniques used to transform data. For more discussions of this topic see especially [1,3,10,19,24].

PROGRAM TRANSFORMATION TECHNIQUES

Renaming: Anti-dependence and output dependence can be removed by using temporaries as discussed above.

IF conversion: Generation of temporary boolean variables to contain results of condition statements. This converts control dependence into data dependence.

Scalar expansion: This technique promotes a scalar variable into an array which often helps to eliminate output dependence. For example, in the converted IF statement above, the variable B was used to hold the value of the conditional expression within the loop. However, since B is a scalar variable, there is an output dependence within the loop. By promoting B to a vector with the range index'range, the loop can now be executed with vector operations:

```
    for i in index'range loop;
        B(i):= conditional expression;
        WHERE (B(i)) A(i) := B(i);
    end loop;
```

As long as the conditional expression can be evaluated in parallel, this loop could now be executed in two vector operations.

Loop fission: This technique breaks a loop into multiple adjacent loops in order to separate data dependencies.

```
    for i in index'range loop
        A(i) := B(i) + C(i);
        D(i) := A(i) * E(i);
    end loop;
```

can be vectorized if broken into the following two loops

```
    for i in index'range loop
        A(i) := B(i) + C(i);
    end loop;
    for i in index'range loop
        D(i) := A(i) * E(i);
    end loop;
```

Loop fusion: Adjacent loops with no data dependencies can be fused into a single loop to reduce vectorization or concurrency overhead.

```
    for i in 1..5 loop
       for j in 1..3 loop
        A(i, j) := B(i, j) + C(i, j);
       end loop;
    end loop;
```

might become

```
    for i in 1..15 loop
        A(i) := B(i) + C(i);
    end loop;
```

Loop interchanging: Since data dependencies may exist only on one of a nested loop indexes, interchanging the loop may remove the dependency on the innermost loop, thus making it a candidate to run in parallel.

```
for i in index'range loop
     for j in subindex'range loop
        A(i, j+1) := A(i, j) * B(i);
     end loop;
end loop;
```

becomes

```
for j in subindex'range loop
     for i in index'range loop
        A(i, j+1) := A(i, j) * B(i);
     end loop;
end loop;
```

where the inner loop can now be vectorized as A(*, j+1) := A(*, j) since there is no recurrence in the ith subscript.

Global analysis: By tracking the values of indexes and subscripts, it is sometimes possible to eliminate data dependencies. For example, in the loop

```
i in 1..N loop
    A(i + K) := A(i);
end loop;
```

if K is not in the range 1..N, no data dependency exists. By keeping track of the value of K wherever such information is available, the compiler can sometimes detect that this is safe to vectorize.

Inter-procedural analysis: Calls within loops or blocks of statements usually prohibit parallelism since it is impossible to know if data dependencies exist between the called procedure and the calling code. Inter-procedural analysis sometimes allows the compiler to know if the called routine modifies any of the data accessed by the calling code.

Global forward substitution: This is used in conjunction with renaming to eliminate non-cyclic data dependencies and in conjunction with global analysis to assist in the tracking of variable values.

Strip mining: This technique creates a nested loop from a single loop. This might be advantageous because of hardware vector register lengths. For example, assuming the hardware vector register length is L,

```
for i in 1..N loop
     A(i) := B(i) + C(i);
end loop;
```

would be transformed into

```
loop1:
for i in 1..N/L loop
     loop2:
     for j in 1..L loop
       A(i, j) := B(i,j) + C(i, j);
     end loop2;
end loop1;
```

The inner loop, loop2, can now be run in vector mode while loop1 is run in concurrent mode.

Loop copying: When embedded conditional statements inhibit parallelism, the compiler can generate two versions of the loop, one which executes in vector mode and one which executes in scalar mode. For example,in the loop

```
for i in 1..N loop
    A(i) := B(i) + A(i + m);
end loop;
```

if m is positive, this loop can be vectorized; if not, there is a data dependency that prohibits vectorization. The following loops could be generated:

```
if m >= 0 then
    loop  -- vector code
          A(i) := B(i) + A(i + m);
    end loop;
else  -- no vector code
     A(i) := B(i) + A(i + m);
end if;
```

Idiom recognition: This technique recognizes common patterns and transforms them appropriately. For example,

```
for i in index'range loop
     S := S + A(i);
end loop;
```

is a sum reduction. A compiler can transform this loop into a call to a library summation routine if available.

FACTORS INHIBITING AUTOMATIC PARALLELISM

Automatic detection of parallelism is has limitations. Some of the factors which interfere with automatic detection are described below.

GLOBAL DATA

Since all of the techniques described in Section 4 focus on reducing data dependencies, it is not surprising that automatic parallelism is generally inhibited by anything that can cause the state of the data to be unknown. A compiler must make the safe choice to execute in serial mode anytime the data is not fully understood. For example, the following statements

```
A := B;
C := D;
```

which appear to contain no data dependencies, cannot be executed in parallel if B and C might reference the same location, or if A and D are aliases for the same location. Thus the use of access variables and aliases can inhibit generation of parallel code. The problem is compounded if A,B,C and D above were array references, since in that case the subscript references might also be aliases for one another.

CALLS

Likewise call statements can preclude parallel code generation since their affects on global data and their parameters might not be well understood.

I/O

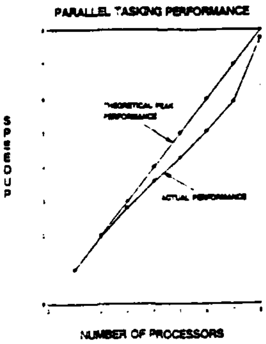
I/O calls usually inhibit parallelism since access to external files usually has to be serialized in order for the output to be meaningful. Additionally many I/O runtime systems are not reentrant.

5. ISSUES SPECIFIC TO ADA

The discussion of parallelism in Ada can be divided into two separate areas. First, because the language defines concurrency at the task level, explicit parallelism at that level is well-understood in Ada. The other levels of parallelism are not explicitly addressed by the language and are therefore less clearly understood and more difficult to achieve. This section will first address issues related to explicit task-level concurrency, and will then discuss issues related to the other levels.

ADA TASKING

Ada's tasks provide an explicit mechanism for task-level parallelism. Task constructs offer benefits to both system design and performance. Unfortunately tasking tends to degrade performance on single-processor machines since these machines must execute both the user code and the additional runtime task management code serially. Multiprocessor machines provide excellent platforms for Ada program execution since they can execute tasks on separate processors. The task-level parallelism defined by Ada can be realized with substantial performance savings on parallel tasking systems. The following graph shows the performance speedup achieved on the Alliant FX/8 using FX/Ada.



The extent of performance enhancement and the degree of code portability are greatly affected by a parallel tasking Ada implementation. An implementation must consider numerous user-level and implementation-level factors.

IMPLEMENTATION-LEVEL FACTORS

Areas that can become bottlenecks if they are not implemented efficiently are memory access, task scheduling, cross-processor synchronization mechanisms, runtime lock granularity, and I/O. Each of these will degrade performance if they are not handled properly.

Memory Access - Globally shared memory available on tightly-coupled multiprocessor machines facilitates a parallel tasking implementation for Ada. Because data can be visible from two or more tasks, and because it is often impossible to know at compile-time whether tasks access shared data, it is important to allocate the entire data space of an Ada program in shared memory. Furthermore, shared memory allows runtime tasking data structures to be efficiently accessed by all tasks in the system.

Task scheduling - Dynamic load-balancing of Ada tasks is essential. The runtime should schedule tasks dynamically to available resources, according to priority and resource requirements.

Cross-processor communication mechanisms - Fast communication mechanisms between processors is essential to implement pre-emptive scheduling and certain language features such as the ABORT statement.

I/O - Since file objects can be shared variables, a parallel tasking system must be able to access the same file from multiple processors. When parallel tasking is implemented with multiple O/S processes, this can be a problem since file descriptors are often not shared between processes.

Runtime Lock Granularity - To achieve scalable performance increases as processors are added to a program, the lock granularity of runtime services must not prevent parallel execution of the runtime. Although some critical sections must be maintained in the runtime to protect its data structures, minimizing use of critical sections is a goal.

USER-LEVEL FACTORS

A parallel tasking Ada system must also consider user-level factors. Among these are:

Portablility - One of Ada's main language design goals was portability. Multi-processor implementations of Ada tasking must not impede portability. Ada code which runs on single-processor machines should execute in parallel on multi-processor machines with no change to the code. Machine details such as the number of processors to use and how tasks are assigned to those processors should be handled in a portable fashion. In fact, these decisions should be deferred to execution time.

Priority - To give the programmer sufficient control over his application, a multi-processor implementation for Ada tasking should support a broad range of priorities enforced by a pre-emptive runtime scheduler.

Software tools - Software tools which help users construct parallel programs using tasking, and which help them debug and profile those programs are important tools in a multi-processor environment.

Parallel tasking systems have proven the performance benefit of Ada tasking. It is important, though, for Ada to support the lower levels of parallelism as well. Tasks should be used where they are appropriate for the design of the application. When used for lower level parallelism, Ada tasks can obscure the design rather than clarify it. Using tasks to implement iterations of a loop to achieve loop-level parallelism, for example, would likely make for awkward design. Furthermore, tasking overhead is fairly expensive. If the task body is too small, the overhead associated with task creation and management might exceed the efficiency gained via parallelism.

ADA AND LOW-LEVEL PARALLELISM

This section will discuss Ada issues related to low-level parallelism. First, problems in this area will be identified and solutions suggested. Problems affecting both explicit and automatic parallelism at this level include exception handling semantics and related code motion restrictions, result accuracy rules, and the semantics of shared composite objects. The lack of predefined vector types and operations affects explicit parallelism. Ada features that promote low-level parallelism will then be discussed. These include range information, global data usage patterns, modes for parameters, and Ada I/O. Finally two features of Ada, pragmas and its approach to environments, facilitate both task-level parallelism and low-level parallelism.

EXCEPTION HANDLING

Exception handling semantics is the major obstacle facing the application of either explicit or automatic low-level parallelism (instruction, expression, loop and block levels) to Ada. According to 11.4.1 and 11.6 of the Ada Language Reference Manual (LRM), exceptions must be raised synchronously with the operation that caused them. If the exception occurrs during evaluation of the RHS of an assignment statement, the store to the LHS of the statement must be abandoned. In any case, control must pass immediately to the appropriate exception handler. According to John Goodenough in AI_00315/11, Section 11.6(4), permits the deferral of raising predefined exceptions such as NUMERIC_ERROR as long as the exception is raised before control leaves the current frame. Even in this case, however, the assignment to the LHS cannot proceed. Since these rules imply an control dependency before every assignment statement that can possibly raise an exception, they interfere with parallel execution.

CODE MOTION

Additionally these semantics severely restrict code motion optimizations. Section 11.6 of the LRM describes legal code motion optimizations. Code motion is allowed only in regard to operand evaluation of predefined operations and in respect to eliminating code whose only purpose is to raise an exception as a side-effect. Reordering assignment statements or changing the order of evaluation of operands of user-defined operations is generally prohibited. Since most of the techniques used to automatically detect and enhance opportunities for low-level parallelism described in Section 4 rely heavily on code motion, Ada's restrictions on code motion significantly impact automatic low-level parallelism.

Even with these rules, there is opportunity for low-level parallelism in Ada. The rules governing exception handling and code motion apply only to cases where there is a visible effect in executing code in a order different from the canonical serial order. Visible effects include either

> raising an exception in the optimized order when no exception would be raised in the canonical order, or
>
> the occurrence of an exception where the data involved with the exception is visible to an appropriate exception handler.

If a compiler can determine that no exception can possibly be raised by a section of code, it is free to apply any optimization to it. Such code can be therefore be executed in parallel mode. Parallel code can also always be generated for operations involving local data that is not referenced in a relevant exception handler. In such cases, there can be no visible effect of executing the code in parallel rather than scalar mode.

Additionally, there is ongoing discussion about a broader interpretation of 11.6 of the LRM among the Ada Language Maintenance Panel and other interested members of the Ada community. Many people have recognized the limitations which Ada's exception handling semantics and section 11.6 have placed upon global optimizations. Ada Issue AI-00315/11 addresses this concern. Having stated that NUMERIC_ERROR is subsumed in CONSTRAINT_ERROR, Ron Brender writing in AI-00315/11 suggests the following rule:

> "It follows that, in the absense of a handler for CONSTRAINT_ERROR, an implementation is allowed to perform all possible optimizations assuming that CONSTRAINT_ERROR will not occur (and need not provide defined behavior if CONSTRAINT_ERROR does occur)."

and states:

> "In the presence of a handler for CONSTRAINT_ERROR, code can be reliably written that depends on the canonical meaning of the program.
>
> In the absence of a handler for CONSTRAINT_ERROR, implementations can provide more optimization than currently allowed (and more easily). For example, pipe-lining can be exploited, vectorization of loops can be employed, and temporaries need not be introduced just to guarantee that an exception precedes rather than follows an assignment."

Given this sentiment and other guidelines proposed by AI-00315/11, full liberty to reorder assignment statements and to store all result values into canonical locations in the process of executing parallel code could be given a compiler if

> 1. The compiler could detect that there was no visible difference in the parallel code execution as compared to the canonical serial execution according to existing rules as described earlier, or
>
> 2. The user directs the compiler to generate parallel code, either with a pragma or compiler switch, and, there is no exception handler for CONSTRAINT_ERROR in the innermost frame for the code being generated.

This solution gives the user complete control while still enabling the the compiler to fully exploit parallelization opportunities. The following sections prove the advantage of this approach.

Instruction-level: Using strict exception handling semantics, instruction scheduling for pipelined architectures becomes much more difficult since stores to canonical locations must be scheduled in such a way as to guarantee that hardware exception delivery precedes the store. Pipeline stalls are invariably introduced to ensure this. When permitted to use the new optimization guidelines, stores to canonical locations can precede exception delivery and pipeline stalls are avoided.

Loop-level: Loop-level parallelism implemented with vector operations is difficult to achieve with current exception handling semantics because a vector operation is atomic. When an exception occurs during a vector operation, the hardware has probably updated all the elements of the result vector before the exception is delivered. (The case where a user vector is processed in a series of vector instructions because the vector length exceeds hardware vector register length is unimportant, since in any case, some number of elements past the element which caused the exception have probably been processed and updated.) Furthermore, the hardware might not report which element of a vector caused the exception, and in the case of multiple exceptions in the same vector operation, it might not report either the first element which caused an exception nor the first exception if more than one kind of exception occurred.

This behavior makes it almost impossible to obey current exception handling semantics since there is no way to avoid updating the element which caused an exception to be raised. The fact that later elements have also been updated would probably be acceptable given the liberty to defer the raising of NUMERIC_ERROR to a later point in the same frame if only those elements not causing any exception were updated.

At first glance, a solution to this problem might appear to be to reexecute a vector instruction in serial mode upon the occurrence of an exception. Such a solution is untenable since the work associated with such reexecution is far beyond the scope of the runtime. Another solution would be the use of vector temporaries. This solution might introduce a performance penalty negating the gain achieved by vectorization.

Using concurrency to implement loop-level parallelism fares a bit better. As long as each processor defers the update of the LHS of an assignment to guarantee that no exception occurred, each processor can execute iterations of the loop in parallel. In this case, even if an exception occurred during the execution of the loop, the behavior of the parallel loop would be the same as the scalar one. Elements of a vector past the point of an exception would be updated, but the element causing the exception would remain unmodified. With these restrictions, this mode would still be useful for loops as long as the stalls necessitated to delay stores did not dominate the code. Using concurrency to implement all forms of loop-level parallelism is not efficient, however, since singly-nested loops with no data dependencies yield the best performance when executed in vector mode. Furthermore, concurrent-outer-vector-inner mode for doubly-nested loops could not be used.

When the broader interpretation of 11.6 as presented above is used, all parallel optimizations could be applied for both vector and concurrent mode. In this case, loop level parallelism for Ada would deliver the same performance benefits as it does for FORTRAN.

Block-level: Block-level concurrency in Ada shares the same restrictions and opportunities for parallelism as concurrency at the loop level as described above.

NUMERICAL RESULT ACCURACY

Numerical results can differ when computed in parallel mode from those computed in scalar mode because of round-off error accumulation order differences. This issue applies to automatic expression-level, loop-level and block-level parallelism. The result value difference may be in violation of 11.6(5) which states that a real result must belong to the result model interval defined for the canonical left-to-right order evaluation of operands. The result difference has been acceptable to scientific and mathematical applications written in FORTRAN. For cases where it is not acceptable, Ada users can avoid low-level parallelism either by not using the compiler option to parallelize, or by specifying an exception handler in the appropriate frame.

SHARED COMPOSITE OBJECTS

The rules for accessing shared variables by two or more tasks are given in Section 9.11 of the LRM. These rules apply to shared variables of scalar or access type. Shared variables of these types can be safely referenced only at synchronization points; that is, a program in which the same scalar or access object is modified by two different tasks outside a synchronization point is erroneous. These rules do not apply to composite types. It is legal for two tasks to simultaneously modify distinct elements of the same array.

It is necessary therefore, to implement vector operations when not applied to the entire vector so that they update only those elements whose values have changed. For example, if task 1 modifies all the even elements of a vector, and task 2 modifies all the odd elements, at the conclusion of these tasks the entire vector has been modified. However, if vector hardware is used, it is likely that each task fetched a copy of the original array, calculated the new values for half the array, and then wrote back their copyof the entire array with only half the elements modified. At the conclusion of the two tasks, the vector would by only half updated by the task that executed the last store instruction. To solve this problem, either an implementation must store back only those elements which were modified, or the language must treat vectors the same as it does scalar objects in terms of shared variables.

EXPLICIT VECTOR CONSTRUCTS

Explicit loop-level parallelism requires vector constructs in the language. Although slices in Ada define contiguous sections of a one-dimensional array, and assignment is defined for arrays, most of the other features presented in Section 3 are lacking in Ada. Ada does not have a predefined vector type. Although implementations can define their own vector types and operations for them, there are advantages to the language providing them instead. First, code using the type would be portable. Currently such code would be portable only to implementations which define the type and its operations in the same way. Secondly, if the language defined a vector type, it could also extend vector referencing expressions to include the expressions described in Section 3. Finally, if it were defined in the language, implementation of vector functionality could be verified and evaluated by the same test suites used for other parts of the language.

ADA ADVANTAGES TO LOW-LEVEL PARALLELISM

Ada incorporates features and software engineering principles that promote automatic detection of low-level parallelism. These include range information, its approach to global data, subprogram specifications, and handling of I/O.

RANGE INFORMATION

Knowledge about subscript values greatly helps a compiler detect opportunities for parallelism. In the example

```
for i in 1..N loop
     A(i+k) := A(i);
end loop;
```

an inhibiting recurrence exists if k is in the range 1..N. In Ada, k's type might have a constraint that ensures the compiler that k will never be in that unsafe range. If the user knew this to be the case, s/he could define a type or subtype for k that would allow the compiler to automatically detect that the parallelization was safe. This method is preferable to FORTRAN's use of a directive explicitly instructing the compiler to generate parallel code because it is more portable and it formalizes the assertion about the values that k can assume.

By propagating range information, an Ada compiler has access to substantial information in regard to the values of subscripts and other data. This information will not only allow the compiler to detect when apparent recurrences are not real recurrences as in the previous example, but it will also help the compiler predict code branches and other behavior. In particular, range information propagation will help the compiler detect when exceptions cannot be raised by certain operations. Eliminating the possibility of exception affords the compiler even more opportunities to generate parallel code.

GLOBAL DATA USAGE

Use of global data interferes with automatic detection of parallelism since a compiler often cannot determine data dependencies when data is global. In these cases, a compiler must make the safe choice not to generate parallel code. The software engineering principles that helped inspire some of Ada's program structure features demand that global data be used sparingly. Ada programs probably use more local data and parameters, and use packages to manipulate what otherwise would be global data structures, than programs written in many other languages. Therefore, the more infrequent use of global data in Ada programs may allow more opportunities for parallelism to be found in Ada.

ACCESS VARIABLE USAGE

The use of access variables is also probably less frequent in Ada than in many languages such as C where pointers are used extensively. Since pointers or access variables also interfere with the data dependency detection, restricted use of access variables promotes parallelism. Ada's strong typing rules encourage programmers to reference objects by name rather than by access variables. For example, C treats arrays and pointers almost identically, leading C programmers to use pointers extensively in array operations. Ada programmers are much more likely to use array references for array operations.

SUBPROGRAM SPECIFICATIONS

Calls to subprograms generally inhibit parallelism since it is usually unknown what side effects the subprogram has on its parameters and global data.Here again, Ada provides an advantage since its parameter passing strategy is much more controlled than that in other languages. First, subprogram specifications are helpful since they specify parameter types. An Ada compiler knows whether an address or value is passed. Furthermore, because parameter modes are specified, an Ada compiler knows if the value of a parameter can change in a called routine. Parameters of mode IN cannot be modified by the called subprogram. Therefore when mode IN is specified for parameters which are not access types, the compiler can assume that the data is not modified. Furthermore, if the subprogram is in the same unit and is not separate, the compiler has access to the entire subprogram body. In this case it can determine how global data is used in the subprogram.

I/O

Unlike other languages where the execution of I/O prohibits parallelism, Ada probably can take advantage of low-level parallelism in the presence of I/O. Because the issue of I/O being conducted in parallel must be solved for the parallel tasking case, it probably does not interfere with low-level parallelism. Parallel tasking systems must already serialize I/O access to the same file. Low-level parallelism with I/O calls will likely serialize the I/O in the runtime.

Ada offers two more advantages to both explicit and automatic parallelism at any level. First it defines the pragma mechanism, and secondly, the Ada environment is a repository of information that can be utilized by any tool.

PRAGMAS

Pragmas can be used to instruct the compiler to generate parallel code for cases where a compiler can not determine that it is safe. For example, a pragma could be used to force a subprogram to be run in parallel with other code, or to inform the compiler that no data dependencies exist where the compiler cannot detect that on its own.

TOOLS

Tools which help the user structure and write code to exploit opportunities for parallelism could be written for Ada. The concept of such a tool fits in very nicely with Ada's programming environment concept, and much of the information which such a tool requires is already available in the program library. Many tools that help programmers design and build programs are already commercially available. Extending these tools to understand parallelism is highly desirable.

6. CONCLUSIONS

Ada can take advantage of explicit and automatic parallelism at all levels. Ada's tasks provide task-level parallelism. Its performance on multiprocessor machines prove the performance benefits of task-level parallelism. Below the task level, Ada's current exception handling semantics sometimes stand in the way of fully exploiting parallelism. The more liberal interpretation of code reordering rules suggested by AI-00315/11 and the approach described here enables Ada to take full advantage of low-level parallelism.

Explicit loop-level parallelism in Ada would be aided by adding a predefined VECTOR type, operations and extended referencing expressions to the language. Even without these, an implementation can provide most of the needed functionality in a supplied vector package thus delivering explicit vector capability to the user.

Ada also includes some unique features which help automatic detection of parallelism. Its range information in particular can help an Ada compiler find opportunities for parallelism that can not be detected in other languages. Ada's limited global data usage and wealth of information stored in program libraries help to reveal additional opportunities.

Modern machine architecture offers vast performance potential through parallelism. It has been shown that Ada can take advantage of parallelism at every level. All forms of parallelism applied in concert to a single Ada program, can yield tremendous performance savings. The challenge is for Ada compiler and runtime technology to make optimum use of these opportunites.

Bibliography

1. Allen, J.R., and Kennedy, K. PFC: A program to convert Fortran to parllel form. Rep. MASC-TR82-6, RIce Univ., Houston, Tex., Mar., 1982.

2. Allen, J.R and Kennedy, K. Automatic loop interchange. In Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction (Montreal, June 17-22). ACM, New York, 1984, pp. 233-246.

3. Allen, J.R, Kennedy, K., Porterfield, C., Warren, J. Conversion of control dependence to data dependence. Conference Record. Tenth ACM Symposium on Principles of Programming Languages. Austin, Texas, January 1983. pp. 177-189.

4. Burke., M. and Cytron, R. Interprocedural dependence analysis and parallelization. Proceedings of the SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN, Not. 21, 7 (July, 1986), 162-175.

5. Chen, Zen and Chang Chih-Chi. Iteration-level parallel execution of DO loops with a reduced set of dependence relations. J. of Parallel and Distributed Computing, 4, 1987, pp. 488-504.

6. Cytron, R.G. Doacross: Byeond vectorization for multiprocessors. In Proceedings of the 1986 International Conference on Parallel Processing (St. Charles, Ill., Aug 19-22). IEEE Press, New York, 1986, pp. 836-844.

7. Cytron, R.G. and Ferrante, J. What's in a Name? -or- The value of renaming for parallelism detection and storage allocation. In Proceedings of the 1987 International Conference on Parallel Processing (Aug. 17-21). Pennsylvania State University Press, University Park, Penn., 1987, pp. 19-27.

8. Cytron, R. Limited processor scheduling of Doacross loops. In Proceedings of the 1987 International Conference on Parallel Processing (Aug. 17-21). Pennsylvania State University Press, University Park, Penn., 1987. pp. 235-242.

9. Dietz, H. and Klapphotz. Refined Fortran: another sequential language for parallel programming. In Proceedings of the 1986 International Conference on Parallel Processing (St. Charles, Ill, Aug. 19-22). IEEE Press, New York, 1986. pp. 184-191.

10. FX/Fortran Programmer's Handbook. Alliant Computer Systems Corporation, Littleton, Ma., 1986.

11. Gibbons, Philip and Muchnik, Steven. Efficient instruction scheduling for a pipelined architecture. Proceedings of the SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Not. 21, 7 (July, 1986), p. 11-16.

12. Hwang, Kai. Advanced parallel processing with supercomputing architecture. Proceedings of the IEEE, 75, 10 (Oct. 1987), pp. 1348-1379.

13. Legal Reorderings of operations, Ada Issue 315 (AI-00315/11), Ada Language Information Clearinghouse Bulletin Board.

14. Kuck, D.J. Parallel processing of ordinary programs. In Advances in Computers, vol. 15, M. Rubinoff and M.C Yovits, Eds. Academic Press, New York, 1976, pp. 119-179.

15. Kuck, D.J. A survey of parallel machine organization and programming. Computing Surveys, 9, 1, Mar. 1977, pp. 29-59.

16. Li, Kuo-Cheng and Schwetman, Herb. Vector C: A vector processing language. J. of Parallel and Distributed Computing, 2, 1985, pp. 132-169.

17. Midkiff, S.P., and Padua, D. A. Compiler generated synchronization for Do loops. In Proceedingts of the 1986 International Conference on Parallel Processing (St. Charles, Ill., Aug. 19-22). IEEE Press, New York, 1986.

18. Padua, D.A., Kuck, D.J., and Lawrie, D.H. High-speed multi-processors and compilation techniques. IEEE Trans. Cmput. C-29, 9 (Sept., 1980), pp. 763-776.

19. Padua, David and Wolfe, Michael. Advanced compiler optimizations for supercomputers. Communications of the ACM. 29, 12 (Dec., 1986)., pp. 1184-1201.

20. Peir, J. K. and Cytron, R. Minimum distance: A method for partitioning recurrences for mulitprocessors. In Proceedings of the 1987 International Conference on Parallel Processing (Aug. 17-21). Pennsylvania State University Press, Univeristy Park, Penn., 1987. pp. 217-225.

21. Polychronopoulos, Kuck, D.J. and Padua, D. Execution of parallel loops on parallel processor systems. In Proceedings of the 1986 International Conference on Parallel Processing (St. Charles, Ill, Aug. 19-22), IEEE Press, New York, 1986, pp. 519-525.

22. Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A-1983). American National Standards Institute, 1983.

23. Sarkar, Vivek and Hennessy, John. Compile-time partioning and scheduling of parallel programs. Proceedings of the SIGPLAN 86 Symposium on Compiler Constructions, SIGPLAN Not., 21, 7, 7 (July 1986), pp. 17-26.

24. Scarborough, R.G., and Kolsky, H.G. A vectorizing Fortran compiler. IBM J. Res. Dev. 30, 2 (Mar. 1986). pp. 163-171.

25. Tang, P. and Yew, P. Processor self-scheduling for multiple-nested parallel loops. In Proceedings of the 1986 International Conference on Parallel Processing (St. Charles, Ill., Aug. 19-22). IEEE Press, New York, 1986, pp. 528-535.

26. Triolet, Remi. Interprocedural analysis for program restructuring with parafrase. Center for Supercomputing Research and Development, Univ. of Ill., Urbana-Champaign, Ill., 1985.

27. Wolfe, M. Advanced loop interchanging. In Proceedings of the 1986 International Conference on Parallel Processing, (St. Charles, Ill, Aug. 19-22), IEEE Press, New York, 1986, pp. 536-543.

The Ada[1] Execution Analyzer - Graphical Support for Multitasking Systems

Anne Clough

The Charles Stark Draper Laboratory, Inc.
Cambridge, Massachusetts

ABSTRACT

Tools that directly support Ada's tasking feature, particularly graphical tools, are inadequate for current Ada applications and will certainly be deficient for the complex applications envisioned for the future. In an internal research and development effort at the C.S. Draper Laboratory(CSDL), a graphical Ada Execution Analyzer (AEA) has been developed to provide the necessary graphical support for the debugging and analysis of Ada tasking applications in a self-target environment. The AEA allows a global view of each Ada task (suspended, ready-to-run, running, rendezvous) as a function of time. This can be very useful in the testing/debugging phase of multitasking system development by showing the exact sequence of events that lead to undesirable behavior in the system. Such support is even more critical in complex embedded, fault-tolerant and distributed systems given their unique requirements and constraints. The need for and impact of such tools must be examined and further research supported so that useful dynamic analysis tools can be developed and provided for the full range of application environments.

INTRODUCTION

In this paper, current techniques for analysis and testing of Ada tasking applications are examined. A graphical approach using timing diagrams is advocated for the testing/debugging phase. Automated tool development to support this approach is then evaluated for the self-target, embedded target, fault-tolerant processor and multi-processor environment.

The paper is organized into four major sections:

OVERVIEW -- Focuses on challenge of testing and analyzing tasking applications and introduces automated timing diagrams, as generated by Draper's Ada Execution Analyzer, as a partial solution that could be applicable to a variety of environments.

TESTING AND ANALYSIS OF ADA TASKING APPLICATIONS -- Surveys current approaches to the problem of testing and analyzing tasking applications.

EXECUTION ANALYSIS USING TIMING DIAGRAMS -- Presents the graphical approach chosen by Draper Laboratory for analysis of tasking applications.

APPLICATIONS -- Discusses four possible environments for such an analysis tool.

Mainframe Environment -- describes the operation of the Ada Execution Analyzer as it is currently used in a VAX 8650 self-target environment.

Embedded Systems -- discusses the requirements that must be met when developing an AEA for an embedded target and describes current efforts at CSDL to accomplish this.

Fault-Tolerant Processors -- discusses extending the tool so that it may be used in a fault- tolerant processor.

Distributed Systems -- presents the need and challenge of providing graphical testing and debugging tools for distributed systems.

OVERVIEW

Incorrect design or implementation of tasking, whether in Ada or any other language which supports parallel processing, will produce unintended task interaction which can degrade system performance and result in deadlock, starvation or incorrect sequencing and synchronization. The Ada community is engaged in a continuing debate concerning correct tasking design. However, automated tools which support the testing of multitask Ada programs are particularly needed today.

The software developer needs to have a means of monitoring how tasks in an Ada system are behaving. Current tools may provide textual information but useful graphical descriptions are not available. For example, traditional debuggers have been augmented so that a programmer can examine the state of the tasks in a system at a breakpoint; however, the larger picture - how tasks operate and communicate over time - is not provided. The user needs a global view of task operation and interaction to understand the incorrect (or unanticipated) sequencing of entry calls and accepts that lead to deadlock, starvation or dead tasks.

In an internal research project at the Charles Stark Draper Laboratory, Inc., an Ada Execution Analyzer (AEA) has been developed. This tool runs as an adjunct to the DEC VAX[2]/Ada debugger and automates the production of timing diagrams. Graphical output (provided as a function of time) shows task states for all tasks in an Ada system , as well as communication between tasks. Use of the tool in the VAX self-target environment has proven its usefulness in quickly identifying how a starvation or deadlock situation has occured. AEA's graphical output usually provides enough insight to correct such problems. Although the primary value of the tool is in the area of analysis of tasking design, its use has also helped Ada students understand tasking concepts much more quickly than would be the case without such a descriptive aid.

Current development is focusing on porting the tool to an embedded real-time target microprocessor environment. Transferring this tool to such an environment has given the development team the opportunity to investigate issues unique to the embedded computer application. A major challenge when providing such a tool for real application development is to assure that the tool does not change the behavior of the system. Extending the use of the execution analyzer to fault-tolerant processor and multiprocessor environments will present even greater challenges. However, as systems become more complex, the need for tools that allow visibility into task interaction becomes more critical.

TESTING AND ANALYSIS OF ADA TASKING APPLICATIONS

Tools are needed by software developers throughout the entire life cycle of a project, from requirements analysis to the maintenance phase. However, most available tools focus on code generation, leaving both early and later phases without support. Testing and analysis tools for Ada applications are particularly needed. Few such tools currently exist; moreover, those available to the software developer have limited usefulness with respect to tasking.

The Ada tasking model continues to be one of the most complex and controversial aspects of Ada. In general, the area of parallel execution is not very well understood, regardless of the implementation language being used. Formal mathematical treatment has proven elusive. Even adopting a standard notation has not been accomplished. The potential for misuse of tasking is great. Given Ada's high-level tasking constructs, the software developer can quickly create very complicated multitasking scenarios. It is much more difficult to have these multitasking systems work properly.

Persistent tasking problems such as deadlock, starvation and incorrect sequencing and synchronization are common. Ideally, the software developer would like to detect these problems before program execution. Toward this end, research continues on static analysis techniques. Unfortunately, the time required to implement such techniques tends to increase exponentially in proportion to the number of tasks in a system, making the practicability of static analysis approaches questionable for large systems.[1] As an alternative to static analysis techniques, some Ada developers recommend severe limitations on the way tasks are used in a system, thereby stripping the Ada language of much of its power.

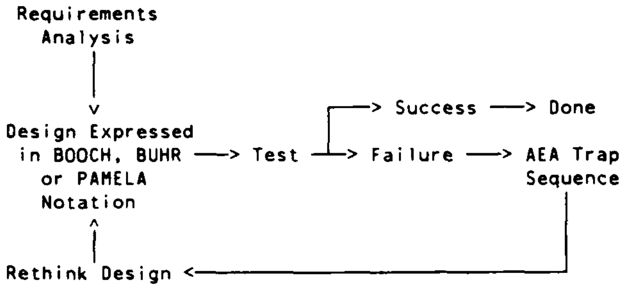
If neither of these "solutions" are acceptable, it appears that, at least for the near future, the need for detecting tasking errors during program execution will remain. Several proposed methods are based on transforming the source program to introduce a new task called the monitor.[2][3] This monitor receives information from all other tasks about their tasking activities and uses this information to detect deadlocks. If the transformed system exhibits deadlocks, it is presumed that the original system would also exhibit deadlocks. There are two objections to this approach: (1) The entire system must be transformed to allow the monitor task to operate, and (2) It is very difficult, if not impossible, to prove that the transformed system is equivalent to the original system. One cannot be certain that deadlocks in the transformed system can be used to accurately predict deadlocks in the original system; similarly, the absence of deadlocking behavior in the transformed system may not preclude its occurance in the actual implementation. In addition, results are dependent on the supporting environment, especially on scheduler characteristics.

The more traditional and less intrusive method of using a debugger to obtain information about a software system is another current testing and analysis option. Most Ada debuggers have been enhanced to provide information about tasks in a system. For example, commands are often provided to enable a user to examine the state of tasks in a system when that system is suspended. It is often possible to set breakpoints or tracepoints when certain tasking states occur and even modify task states during program execution.[4] However, the user is often overwhelmed with textual detail and still does not have a clear picture of how tasks are operating in the system.

EXECUTION ANALYSIS USING TIMING DIAGRAMS

Some type of graphical support for tasking applications is clearly needed. The timing diagram concept was chosen because it is an easily understood and useful way to visualize task states and task interaction over the lifetime of a system. As a part of the internal research effort, a methodology was formulated for using timing diagrams throughout the development process to ensure that the tasking behavior of programs is specified correctly and executes as specified.[5]

Standard notations (Booch diagrams[6], Buhr notation[7], PAMELA[3][8]) used by Ada developers in the design phase of a project reflect the designers view of the operation of a system. Because there is no assurance that the system will work over all permutations of inputs, timing and interrupt sequences, and because there is no closed form theory for the stability of such a system for all possible inputs, such systems can fail. Using the AEA during the testing phase shows how a system is operating for a specific sequence of inputs. Once the AEA traps an unanticipated sequence of inputs that causes problems, the developer can then go back to the original design, apply that sequence and discover any logic errors. Use of the AEA in the design/test/redesign cycle is shown below:



Software developers can gain insight into how tasks execute and often important information about why their design has failed. The effect of various language constructs can be gauged. In general, such a tool facilitates better understanding of the critical aspects of complex multiprograms. Graphical time-line outputs can certainly reduce pages of textual information into a concise and useful pictorial form. To use such debugger output to reconstruct task interaction in a software system would be tedious and error-prone. Clearly, an automated graphical capability is invaluable. Ada Execution Analyzer output is worth hundreds of pages of debugging printout.

APPLICATIONS

Mainframe Environment

The Ada Execution Analyzer (AEA), developed under a research and development grant at the C.S. Draper Laboratories in 1986, automates the production of timing diagrams following program execution. Task states (running, suspended, ready-to-run/active, inactive) and task interactions are clearly shown. Information about each task in a system is logged every time a task switch or task rendezvous occurs. This information is displayed graphically as a function of time, thereby giving the software developer a true picture of what is going on in a system. The AEA runs as adjunct to the DEC VAXAda debugger and, as such, uses debugger commands to log information about task states at each task switch.

The AEA produces two kinds of timing diagrams that help to visualize Ada task interaction:

(1) An overview diagram which displays up to 20 Ada tasks (see Figure 1).
(2) A detailed diagram which displays up to 5 Ada tasks (see Figure 2).

Symbols are defined in Tables 1 and 2. Table 1 contains the task states and Table 2 contains the task substates. Since the Ada Execution Analyzer runs as an adjunct of the DEC VAX debugger, information shown on the timing diagrams and explained in these symbol tables reflects information on task states available from the debugger.[9]

Embedded Systems

The Ada Execution Analyzer operates in a relatively simple configuration where compilation, execution, debugging, and run-time analysis occur in the host processor/operating environment. However, testing and analysis tools for more complex environments, such as embedded systems, are critically needed. An Embedded Ada Execution Analyzer (EAEA), currently under development at Draper Laboratory, will provide graphic execution traces of system events in real-time on a target microprocessor.

Developing a useful testing and analysis tool for embedded applications requires that developers pay attention to two unique requirements of such applications:

(1) The Ada code that runs on the target during testing and analysis must be identical to the code that will be delivered.

(2) The tool itself must have at worst a negligible effect - at best, no effect - on the run-time performance of the entire system.[10]

The first requirement effectively eliminates any testing approach that requires instrumented code or the introduction of testing or monitoring tasks. Compiler designers have attempted to answer the second requirement by developing host-target debuggers for embedded targets.[11][12] Source level debugging of code running on the target microprocessor can be provided by keeping all the Ada-related functions (Ada source code, symbolic information available in the Ada program libraries, compiler and builder symbol tables) on the host. A target debug monitor is then installed on the target microprocessor to provide the necessary low level support, such as reading and writing target memory, setting breakpoints, downloading and running application programs. Such a monitor will have a negligible effect on run-time performance as it will not affect program execution until it is explicitly directed to intervene. The monitor itself is very small and only requires a small amount of target memory. An alternative approach uses a hardware emulator to provide even less debugger intrusion.

CSDL's EAEA runs as an adjunct to the Verdix VADS[4] VAX to M680X0 debugger; it provides graphic execution traces of system events in real-time on the target processor. The Verdix target debug monitor (TDM), installed on a M68020 microprocessor, is used to extract the necessary information about task states and communication during program execution. Uninstrumented code is used. Relinking is not necessary to use the debugger. Therefore, program execution, in terms of task sequencing and interaction, should not be affected if the tool is operative during a run. The TDM will intervene only when invoked. When this happens at task switches and rendezvous, the system will be suspended and restarted when the logging activity is complete. The entire run will take longer, but logging output will reflect the real system's sequencing and interaction. Important diagnostic information about the system will be supplied on the system trace, which shows events in the system on a relative (rather than absolute) time scale.

Fault-Tolerant Processors

The EAEA can also supply important system trace information for the fault- tolerant processor. In CSDL fault-tolerant architectures,[13] processors run in synchrony, using redundancy to provide back-up computing power in the event of processor failure.[14] A target debug monitor could be installed on each processor in such a configuration. A debugger interface on the host would handle the outputs being received from each processor. These debugger outputs should be identical; if they are not, the output which is not in agreement with the output of the other two processors will be rejected. In this way, disparities in processor operation will be handled as they are throughout a fault-tolerant system.

A simpler interim approach would eliminate the need to provide the interface described above by providing debugging capabilities on one node of a fault- tolerant processor at a time. Since processors are running redundant code, a trace produced by one processor should mirror traces produced by the other processors in the configuration. If lack of true synchrony is suspected, a developer could request traces to be run separately, using debugging capabilities on each processor in the fault-tolerant configuration in turn. Results could then be compared.

In either case, it is essential that synchronous operations be preserved. The action of any debugger or tool must not disrupt or adversely affect the synchronous operation of the fault-tolerant processor. If the EAEA wishes to collect information about task states at every task switch, it is essential that all processors in the configuration be stopped synchronously when this logging activity takes place and restarted synchronously when it is completed. If the target debug monitor can be modified to stop the clock when breakpoints are reached, absolute timing information could be added to the relative timing information now provided by the tool.

The first requirement, guaranteeing that synchronous operation is preserved, is the most difficult. Hardware solutions to provide this are being investigated. The second concern, disabling the clock whenever a breakpoint is reached, can be addressed by modifying trap instructions within the target debug monitor to provide trap handlers to disable the clock.

Distributed Systems

There is undeniably a need for a tool that will help the software developer analyze task operations and interactions in a single processor or fault-tolerant configuration. An even more pressing need exists for a tool that will enable the developer to trace synchronization and communication events for applications that share multiple machines. Certainly, the possibility of errors in task sequencing and interaction increases dramatically with the complexity of the system. Working on a distributed system, without the benefit of easily tracking task interactions across processors and throughout the system, is unacceptable, both in terms of personnel productivity and in terms of the quality of the resulting software.

An Ada Network Execution Analyzer would graphically show the behavior and interaction of tasks across a network. To accomplish this, it is necessary to log task switching/rendezvous information from each processor in the network. In addition, it is necessary to log task calls and rendezvous between processors. If each switch and rendezvous is time-tagged, perhaps by using the system clock, then the switching/rendezvous information for all tasks in the system could be sorted at the end of a run and timing diagrams could be generated.

The feasibility and cost of developing such a tool is currently being studied. As with AEA tools already under development, it is of paramount importance that any network analyze· tool not change the way a distributed system operates. However, stopping an entire distributed system while information is being logged may not be either desirable or possible. Consequently, other approaches, such as building a logging facility into the run-time system or using a trace/replay approach[15], are being examined.

CONCLUSION

The need for analysis tools of all types, including graphical tools, is acute. Ada applications, particularly those utilizing tasking, are not well served by existing tools.

Through the research and development effort currently underway at Draper Labs, the potential for developing useful graphical tools has been demonstrated. Further research is necessary, particularly in the area of fault-tolerant processors and distributed systems. Such research should be supported and rigorously pursued so that software developers can have the tools that are needed to develop the complex applications that are envisioned for Ada.

ACKNOWLEDGEMENTS

The original idea of using timing diagrams to analyze tasking systems was suggested by Professor Richard Vidale of Boston University. Development of the Ada Execution Analyzer was carried out by Paul Szulewski, Richard Vidale and Judith Weiss. The author would also like to thank Stuart Schiffman and Connie Serra for their work in adapting the Ada Execution Analyzer for use with embedded systems.

The support and advice of Bob Brown, Sid David, Brooke Green and Avram Tetewsky of CSDL has been particularly appreciated in preparing this paper.

REFERENCES

[1] Taylor, R., "A General Purpose Algorithm for Analyzing Concurrent Programs," Communications of the ACM, Vol. 26, No. 5, May 1983, pp. 362-376.

[2] German, S., "Monitoring for Deadlocks and Blocking in Ada Tasking," IEEE Transactions on Software Engineering, Vol. SE-10, No. 6, November 1984.

[3] Helmbold, D. and Luckham, D., "Debugging Ada Tasking Programs," IEEE Software, March 1985, pp. 45-57.

[4] Conti, R.A., "Debugging Ada Tasking Programs," Annual National Conference on Ada Technology, 1985, pp. 72-81.

[5] Vidale, R.F., Szulewski, P.A. and Weiss, J.B., "Visualization, Design and Verification of Ada Tasking Using Timing Diagrams," Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station, Houston, Texas, June 1986, pp. D.3.2.1-D.3.2.11.

[6] Booch, G., Software Engineering with Ada, second edition, Benjamin/Cummings Publishing Co., Inc., 1987.

[7] Buhr, R.J.A., System Design with Ada, Prentice-Hall, 1984.

[8] Cherry, G.W., PAMELA Designer's Handbook, Thought Tools, Inc., Reston VA, 1986.

[9] Developing Ada Programs on VAX/VMS, Digital Equipment Corporation, Maynard, MA, February 1985, pp. 8-12 to 8-14.

[10] Collingbourne, L.R., "A Practical Approach to Developing Real-Time Ada Programs for Embedded Systems," Proceedings of the International Workshop on Real-Time Ada Issues, May 1987, pp. 15-17.

[11] Domitz, R.O., "Real-Time Ada Debugging," Proceedings of the International Workshop on Real-Time Ada Issues, May 1987, pp. 18-20.

[12] Burns, G., "Cross-Debugging Real-Time Ada Programs," Proceedings of the International Workshop on Real-Time Ada Issues, May 1987, pp. 21-23.

[13] Hopkins, A.L., Lala, J.H. and Smith, T.B., "The Evolution of Fault Tolerant Computing at The Charles Stark Draper Laboratory, Inc., 1955-85", Dependable Computing and Fault Tolerant Systems, Vol. I: The Evolution of Fault-Tolerant Computing, ISBN-0-387-81941-x, pp. 121-140, Springer-Verlag, Wien, Austria, 1987.

[14] Lala, J.H., "A Byzantine Resilient Fault-Tolerant Computer for Nuclear Power Plant Applications", Digest of Papers FTCS-16: The 16th Annual International Symposium on Fault Tolerant Computing, Vienna, Austria, July 1986.

[15] Hutcheon, A.D., Snowden, D.S. and Wellings, A.J., "Programming and Debugging Distributed Real-Time Applications in Ada," Proceedings of the International Workshop on Real-Time Ada Issues, May 1987, pp. 73-75.

AUTHOR

Anne Clough, staff engineer - Ada Office, has been with C.S. Draper Laboratory since 1984. As a staff member in the Ada Office, she has been actively involved in introducing Ada technology to CSDL. Toward this end, she has developed and taught courses in the Ada language and software engineering, participated in Ada projects, been involved in the benchmarking of Ada tools and is presently principal investigator of an internal research and development effort directed at developing an integrated software development environment. Ms. Clough has a BS degree in mathematics from the University of Massachusetts, an MS in mathematics from University of Lowell and an MS in computer engineering from Boston University.

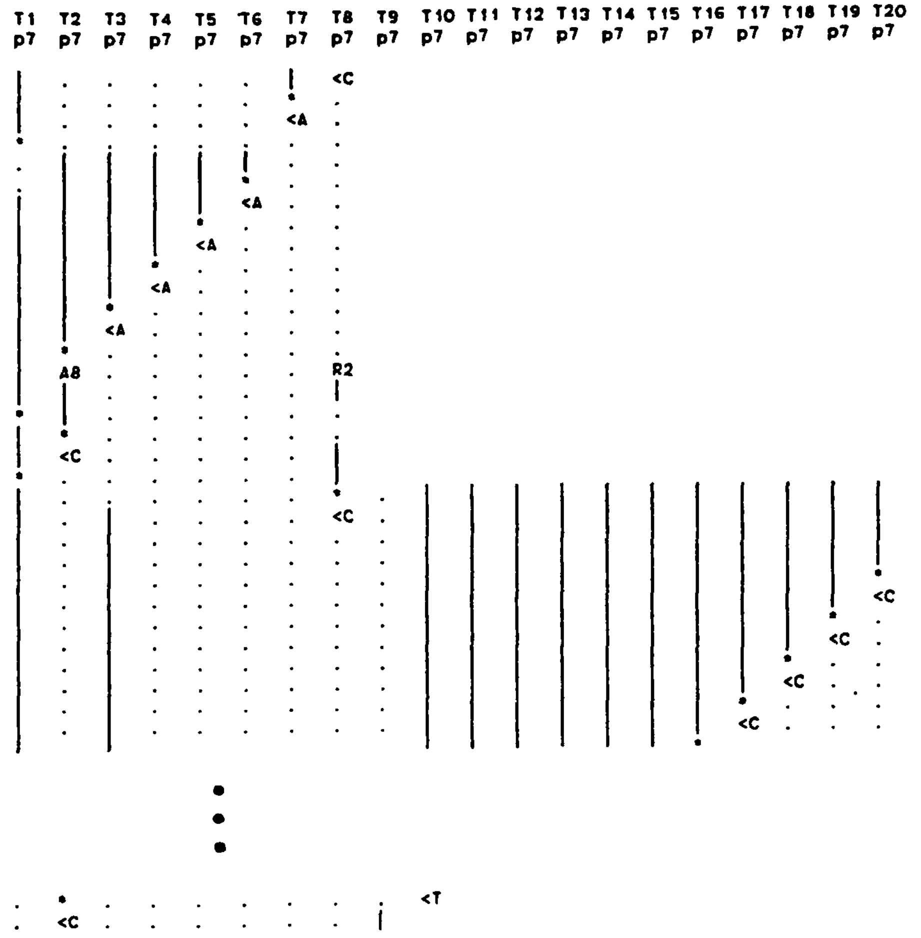
The Charles Stark Draper Laboratory, Inc.
555 Technology Square
Cambridge, Massachusetts 02139

[1] Ada is a registered trademark of the Department of Defense (Ada Joint Program Office).

[2] VAX is a trademark of Digital Equipment Corporation.

[3] PAMELA is a trademark of George W. Cherry.

[4] VADS is a trademark of Verdix Corporation.


T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12 T13 T14 T15 T16 T17 T18 T19 T20
p7 p7 p7 p7 p7 p7 p7 p7 p7 p7 p7 p7 p7 p7 p7 p7 p7 p7 p7 p7
<C
<A
<A
<A
<A
<A
A8
R2
<C
<C
<C
<C
<C
<C
<C
<T
<C

```
1   7   87924
2   7   BANK.TELLER(1)
3   7   BANK.TELLER(2)
4   7   BANK.TELLER(3)
5   7   BANK.TELLER(4)
6   7   BANK.TELLER(5)
7   7   BANK.TELLER(6)
8   7   BANK.DISPATCHER
9   7   MAIN.CUSTOMER(1)
10  7   MAIN.CUSTOMER(2)
11  7   MAIN.CUSTOMER(3)
12  7   MAIN.CUSTOMER(4)
13  7   MAIN.CUSTOMER(5)
14  7   MAIN.CUSTOMER(6)
15  7   MAIN.CUSTOMER(7)
16  7   MAIN.CUSTOMER(8)
17  7   MAIN.CUSTOMER(9)
18  7   MAIN.CUSTOMER(10)
19  7   MAIN.CUSTOMER(11)
20  7   MAIN.CUSTOMER(12)
```

Figure 1. AEA Overview Diagram

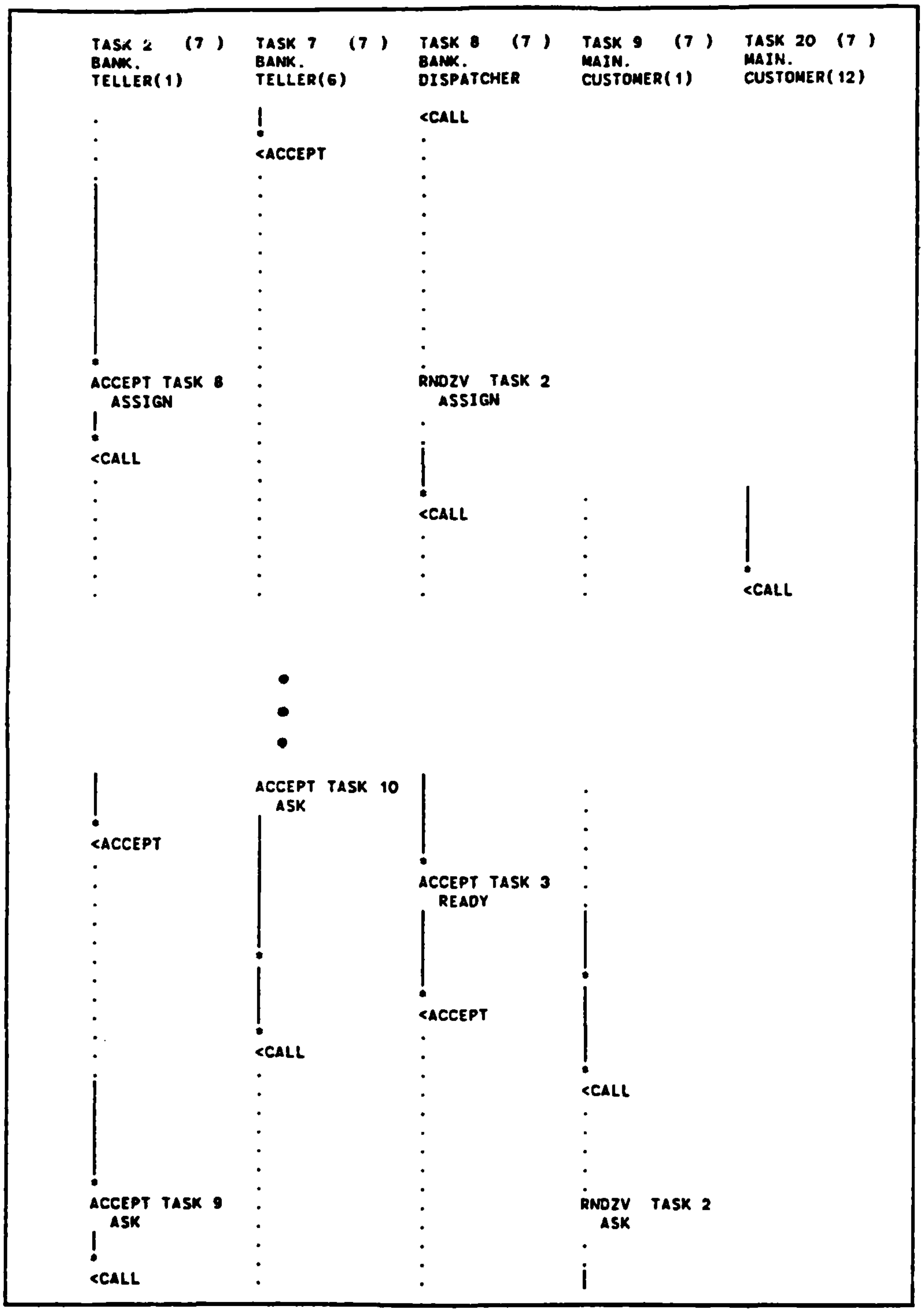


Figure 2. AEA Detailed Timing Diagram

Table 1. AEA Overview and Detailed Diagram States

	TASK STATES	
TIMING DIAGRAM SYMBOLS	OVERVIEW DIAGRAM SYMBOLS	MEANING
TASK N (#)	TN P#	Task number N with priority #
UNIT.TASK_NAME		Logical name of program unit that declares TASK_ NAME
		POINTS OF RENDEZVOUS:
RNDZV TASK #	R#	Task has rendezvoused with task #
ENTRY_NAME		Task #.ENTRY_NAME
ACCEPT TASK #	A#	Task has accepted call from task #
ENTRY_NAME		Accept ENTRY_NAME
		TASK STATES:
* *	* *	Task is running
|	|	Task is ready to run
. .	. .	Task is suspended
<TERM	<T	Task has terminated

Table 2. AEA Overview and Detailed Diagram Substates (Part 1 of 2)

TIMING DIAGRAM SYMBOLS	OVERVIEW DIAGRAM SYMBOLS	TASK SUBSTATE	MEANING
<ABORT	<AB	Abnormal	Task has been aborted.
<ACCEPT	<A	Accept	Task is waiting at an accept statement that is not inside a select statement.
<Completed[ab	<CA	Completed[abn]	Task is completed due to an abort statement, but is not yet terminated. In Ada, a task awaiting dependent tasks at its "end" is called "completed". After the dependent tasks are terminated the state changes to terminated.
<Completed[ex	<CE	Completed[exc]	Task is completed due to an unhandled exception, but is not yet terminated. In Ada, a task awaiting dependent tasks at its "end" is called "completed". After the dependent tasks are terminated, the state changes to terminated.
<Completed	<CO	Completed	Task is completed. No abort statement was issued, and no unhandled exception occured.
<Delay	<DL	Delay	Task is waiting at a delay statement.
<Dependents	<DP	Dependents	Task is waiting for dependent tasks to terminate.

Table 2. AEA Overview and Detailed Diagram Substates (Part 2 of 2)

TIMING	OVERVIEW		
DIAGRAM	DIAGRAM		
SYMBOLS	SYMBOLS	TASK SUBSTATE	MEANING
<Dependents[e	<DE	Dependents[exc]	Task is waiting for dependent tasks to allow an unhandled exception to propagate.
<CALL	<C	Entry call	Task is waiting for its entry call to be accepted.
<Invalid State	<IV	Invalid state	There is a bug in the VAX Ada run-time library.
<I/O or AST	<IO	I/O or AST	Task is waiting for I/O completion or some AST. (Asynchronous system true).
<Select or del	<SD	Select or delay	Task is waiting at a select statement with a delay alternative.
<Select or Ter	<ST	Select or term.	Task is waiting at a select statement with a terminate alternative.
<SELECT	<S	Select	Task is waiting at a select statement with neither an else, delay, or terminate alternative.
<Shared resour	<SR	Shared resource	Task is waiting for an internal shared resource.
<Terminated[a	<TA	Terminated[abn]	Task was terminated by an abort.
<Terminated[e	<TE	Terminated[exc]	Task was terminated because of an uhandled exception.
<Terminated	<TN	Terminated	Task terminated normally.
<Timed entry	<TI	Timed entry call	Task is waiting in a timed entry call.

The Application of Ada To a VHSIC Hardware Embedded System

Robert J. Abel
Cindy C. Dare
Richard J. Prokop
William R. Snow

ESL Incorporated Sunnyvale, CA

ABSTRACT

Although the Ada language was developed for real-time embedded applications, very little work to date has reported on the use of Ada in these applications. The VHSIC TAM system utilizes the unique combination of VHSIC hardware and Ada software technologies to attack a demanding embedded application: Threat Association used in Electronic Warfare. VHSIC TAM is a distributed system making use of both a workstation and a multi-processor embedded computer. Design of the system utilized concepts from Object-Oriented Design. Attention was paid to Software Reuse. This paper relates the experiences encountered during development that are directly related to the use of Ada.

INTRODUCTION

Although the Ada language was developed for real-time embedded applications, very little published work to date has reported on the use of Ada in real-time embedded systems. Much of the reported work has focused on general purpose computing problems that are typically solved on a general purpose computer running a general purpose operating system. More specifically, little work has been reported on the use of Ada in Electronic Warfare (EW) Applications. Some work has been reported in other real-time areas (such as avionics [1]), however, the real-time constraints are typically less stringent in these areas. This paper reports on the use of Ada in the area of Electronic Warfare signal processing; this area has historically been known to be one of the most demanding real-time embedded applications.

Threat Association is a key function of many EW systems. The threat association function compares characteristics of intercepted emissions with those of known signals. Threat Association facilitates identification of the type of emitter and type of platform from which the signal originated. Modern scenarios require that associations be performed in real time at a very high rate.

To meet the challenge of the threat association problem, ESL developed the VHSIC TAM (Very High Speed Integrated Circuit Threat Association Module). VHSIC TAM is a high performance system that implements the real-time threat association function, as well as a user interface and environment simulation capability. VHSIC TAM attains the necessary processing power by combining two new technologies: VHSIC hardware and Ada software. It also utilizes recent design methods, and strives for software reuse.

VHSIC TAM was developed under contract from the United States Army's Electronic Warfare Reconnaissance Surveillance and Target Acquisition Center (EW/RSTA), Fort Monmouth, New Jersey[1]. The system was completed in October, 1987, and a follow-on effort is currently in progress.

This paper examines the use of Ada in the VHSIC TAM system. It not only describes the Ada software architecture but also relates the developers' experiences using Ada. The intent of the paper is to document those issues encountered in development of a real embedded application which are directly related to the use of Ada. The intent is not to simply provide a list of Ada's strong and weak points. The paper begins with an extended problem statement followed by an overview of the system architecture. It then describes the Ada

[1] VHSIC TAM is a VHSIC insertion program for the MEDFLI system.

software design methods used, the system software design and important issues and experiences. The paper concludes with a list of our observations and insights.

PROBLEM STATEMENT

A particularly difficult threat association problem is posed by a "dense" signal environment. In these environments, a large amount of data must be processed and analyzed. The situation is made more difficult when the *instantaneous collection bandwidth*[1] is maximized to provide high probability of intercept. A wide instantaneous bandwidth requires the EW system to process many signals simultaneously. Typically, an EW system must process hundreds of signals per second and report the results within milliseconds.

Before data collected by the EW system reaches the VHSIC TAM unit, the data is preprocessed by a signal sorting function that separates the signals within the instantaneous collection bandwidth. Thus, the input that VHSIC TAM receives from the system "front-end" is a stream of reports that characterize the received signals according to the signal parameters measured. The threat association function uses these reports to unambiguously associate an identification to each report. Association entails a complex processing procedure; central to the procedure is a comparison between the parameters of the received reports and those of a resident database. These comparisons represent a large part of the threat association processing burden. Parametric overlap between measured signals and the resident database indicate multiple potential source identities. Multiple identities may then be further resolved by application of additional algorithms.

SYSTEM ARCHITECTURE

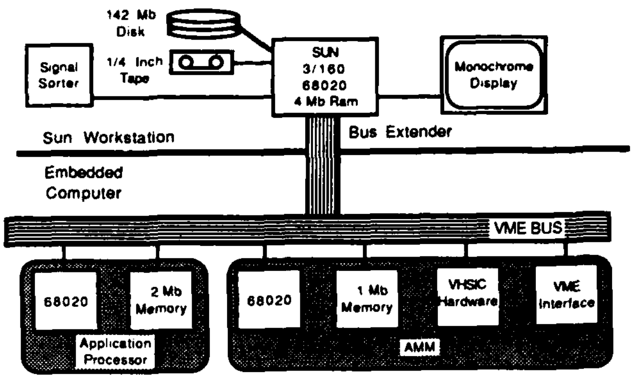


Figure 1: VHSIC TAM Hardware Architecture

The VHSIC TAM architecture is a distributed system consisting of a multiprocessor embedded computer and a workstation. The hardware architecture is shown in Figure 1. The embedded computer hosts the real-time threat association function. The workstation hosts the Man-Machine Interface (MMI) and an environment simulator in addition to interfacing to the signal sorter. The MMI provides the operator with system control and displays the results of the threat association. The environment simulator generates "front-end" reports that are processed by the threat association function. It provides standalone testing of the system.

The multiprocessor embedded computer is a high performance parallel processing unit that is intended to be located on a collection platform. The unit consists of a Motorola 68020 application processor with memory board, and a special purpose Associative Memory Module (AMM) interconnected by a VME backplane[1]. The AMM is comprised of a 68020 control processor, memory board, VME interface, and the VHSIC hardware. Collectively, the units of the AMM implement a high speed search processing unit which performs the parametric comparisons between reports and a resident database. The performance of the AMM is achieved through the use of a VHSIC Window Addressable Memory (WAM) array. WAMs are capable of performing multidimensional comparisons at an extremely fast rate. The AMM control processor executes an assembly language program. The application processor executes an Ada application program and runtime environment.

The workstation is a Sun Microsystems Sun 3.[2] This

[1] Receivers collect large amounts of data in a very short period of time.

[1] The AMM hardware and its controlling software were developed under subcontract to TRW MEAD.

[2] Specifically, we chose the Sun 3/160. This workstation features a 68020 processor running at 16 MHz, monochrome bit-mapped display, and a VME backplane. Our configuration had 4 Megabytes of memory. The workstation runs UNIX 4.2BSD as its operating system.

workstation features the UNIX[1] operating system and a bit-mapped graphics display. All software that executes on the workstation is programmed in Ada, with the exception of required interfaces to UNIX[2], which are programmed in C.

Both the Sun and the embedded computer are VME based systems. A commercially available VME bus extender provides the high speed interface required between the workstation and the embedded computer.

The VHSIC TAM architecture represents the future in high performance real-time signal processing and user interface systems. Multiprocessor embedded computers augmented with high speed special purpose coprocessors to perform burdensome computational tasks, and software implemented in the Ada language, can be expected to dominate future military signal processing systems. Operator workstations can also be expected to dominate future systems because window-based graphical user interfaces provide unmatched flexibility and cost effectiveness.

DESIGN METHODS

An initial step involved selection of a development environment and design methods suitable for the VHSIC TAM project. An appropriate development environment was a host-target configuration. Code for both the workstation and embedded computer was developed on the workstation. Code for the embedded computer was cross-compiled on the workstation and downloaded to the embedded computer. VHSIC TAM used the Verdix Ada Development System (VADS) to do the software development. VADS provides the Sun native compiler in addition to the 68020 cross-compiler[3].

Two design methods were studied: Functional Decomposition and Object-Oriented Design (OOD) [2],[3]. Historically, Functional Decomposition has been widely used by many companies, including ESL. This was probably a result of the number of projects using FORTRAN; Functional Decomposition is quite effective for design of FORTRAN programs.

For some time, ESL has researched alternate software design methods. This research is motivated partially by the desire to build complex software systems (such as distributed systems), and partially by the acceptance of powerful languages such as Ada. Ada fueled our interest in Object-Oriented Design. Object-Oriented Design supports software engineering concepts not directly addressed by Functional Decomposition; two examples include concurrency and information hiding. These concepts proved to be helpful when designing a complex system like VHSIC TAM.

Each method has distinct advantages and disadvantages. Functional Decomposition is a commonly used and well-understood method. However, it lacked support for the notion of concurrency within a system. Object-Oriented Design supports the notion of concurrency, but this method was relatively new to us and its use was not without risk. The developers further researched Object-Oriented Design. Direct application of this method on VHSIC TAM was not acceptable as it would have proven too tedious and would have caused a proliferation in the number of objects and tasks in the system. Unnecessary tasking overhead was not acceptable for this system. Additionally, direct application would have been difficult because the system specifications were incomplete.

The best approach for the design was to use a combination of the two methods. The system was functionally decomposed into subsystems. Each subsystem was then decomposed into modules[1]. Many modules were separate threads[2] of execution. Module designs incorporated the main concepts of Object-Oriented Design.

The Choice To Use Ada

Ada is the language chosen for VHSIC TAM. This was not a clear choice when the program began. VHSIC TAM required a high-level language combined with 68020 assembler. The high-level candidates included Ada, C, and Pascal. The developers had the most

[1] UNIX is a registered trademark of AT&T.

[2] It was necessary to interface to UNIX in order to access devices, low level graphics primitives, and to interface to C routines that could not be implemented in Ada (bitwise logical operations for instance).

[3] The project was developed using version 5.3 of the native compiler and an engineering version of the cross-compiler.

[1] A module is a self-contained, logical piece of the design. It has a small set of well-defined functions to perform and a well-defined interface.

[2] A thread of execution refers to a logical flow of execution in a program. Tasks provide a way to implement threads of control.

experience with C; they had the least experience with Ada. The predominant experience with C ruled out Pascal, leaving the choice between Ada and C. C had the advantage of minimizing risk; all of the developers had completed substantial programs in C and programs similar to VHSIC TAM had been delivered using C. On the other hand, the use of C could not provide the additional benefits that the use of Ada could: First, the Department of Defense has made a substantial investment in Ada and is anxious to reap benefits from this investment. Second, the customer was technology-oriented and interested in the benefits provided by Ada. Third, use of Ada would allow ESL to gain a competitive advantage.

But we perceived Ada to be risky; none of the developers had done substantial Ada programming and ESL had never delivered a program (much less one employing an embedded computer) that used Ada. The industry-wide programming community harbored vociferous Ada skeptics. Compilers were immature and good runtime environments for embedded applications were virtually non-existent. However, if the project could be completed successfully using Ada, the benefits to both the customer and to ESL would be enormous. The customer would have experience with and insight into the development and performance of a complex Ada-based system. ESL understood that many future contracts would list Ada as a requirement. Additionally, ESL would understand more about this new technology, enhancing ESL's ability to deliver state-of-the-art products to the customer and ESL's competitive advantage in the industry.

Ada was chosen after gaining customer support. To mitigate risk, Ada and Ada development products were investigated. In addition, a contingency plan was created that would allow program completion using an alternate language if Ada was found to be too immature. And even if Ada was too immature, an important contribution would still be made to the customer because there would be concrete evidence of the ways in which Ada is risky. This information would serve as valuable input to future programs considering the use of Ada.

SYSTEM SOFTWARE DESIGN

The top-down design of the system software was begun using Functional Decomposition to decompose the system into four subsystems: the Control, Data Input, Information, and Threat Association subsystems. Each subsystem was then further decomposed into its constituent modules using Object-Oriented Design concepts. The modules mapped into Ada package specifications. The final software design and the allocation of the software to the hardware is shown in Figure 2. The Data Input and Information subsystems are resident entirely on the workstation and the Threat Association Subsystem is resident entirely on the embedded computer. The control subsystem spans both computers.

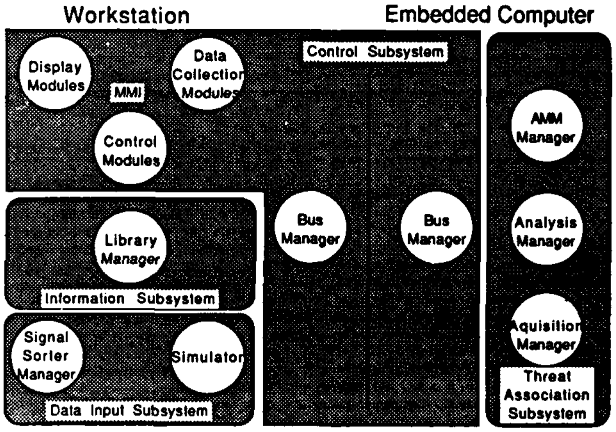


Figure 2: VHSIC TAM Software Subsystems and Modules

Software Subsystem Overview

Control Subsystem

The control subsystem is composed of the MMI and the Bus Managers. The MMI is the operator's window into the system and is the largest portion of this subsystem. The MMI module is broken down into three submodules: control modules, display modules, and data collection modules. Control modules handle device input and manage the allocation of display modules to areas of the screen. Display modules manage windows. Each window is an instance of a tactical display such as the current system configuration or the threat association results. Data collection modules collect data from input devices and often serve as buffers between devices and display modules. The Bus Managers control the bus extender and enable the embedded computer and the workstation to exchange large quantities of data.

Data Input Subsystem

The data input subsystem is the entry point for intercept reports processed by the system. Normally, intercept reports are supplied by a signal sorter. The Signal Sorter Manager handles all aspects of this interface; it provides for signal sorter control and for intercept report acquisition. In absence of a signal sorter, intercept reports are supplied by the environment simulator. This module simulates both the operational environment and the signal sorter interface.

Information Subsystem

The Information Subsystem consists of the parametric databases and configuration information used when running the system. A parametric database drives the threat association function. The library manager is the main module in this subsystem. It handles the creation and maintenance of this database.

Threat Association Subsystem

The threat association subsystem performs the threat association function for the system. It consists of three main modules: the acquisition manager, the analysis manager, and the AMM manager.

As reports are received from the workstation over the bus, they are collected by the acquisition manager. The acquisition manager bundles intercept reports into groups sized to make efficient use of the AMM.

The raw reports received by the acquisition manager have to be analyzed to identify them. The analysis manager initiates searches through the parametric library to get a list of possible candidate identifications for each emitter report. It then performs other algorithms to reconcile any ambiguities. It sends the final results to the operator.

The AMM manager provides a "server" interface to the VHSIC search hardware. The analysis manager requests searches through its interface.

SYSTEM DATA FLOW

The major data flow through these modules is depicted in Figure 3. Intercept reports originate at the Simulator or Signal Sorter Manager. These reports are sent to the embedded computer through the bus managers. The acquisition manager collects the intercepts, bundles them, and hands them off to the analysis manager. The analysis manager sends the intercepts through searches initiated by the AMM manager, runs analysis algorithms on the results, and sends the results back to the workstation through the bus managers. A data collection task on the workstation receives the results and passes them on to a display task if one exists.

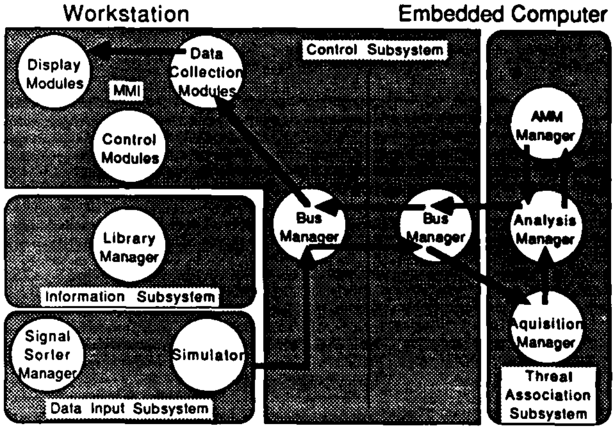


Figure 3: VHSIC TAM Report Processing Data Flow

ADA-RELATED DESIGN DECISIONS

It is instructive to take a closer look at some of the design decisions that went into several of the modules with the intent of highlighting the design decisions that could be directly associated with the choice to use Ada. Ada simplified some decisions while complicating others.

The Man-Machine Interface

MMI Use of the Ada Tasking Model

The VHSIC TAM MMI is a multi-window bit-mapped graphical display. It displays any combination of text, vector graphics and imagery in a window. MMI input devices include the mouse, keyboard, and bus extender. Other devices are easily interfaced to the MMI.

The MMI design took advantage of the Ada tasking model to create a modular windowing system. The design took advantage of the fact that the MMI is logically many different threads of execution; there are threads for handling input devices, managing the screen, collecting data, and for drawing displays. The design translated the input handling and screen management threads into control tasks, data collection threads into data collection tasks, and display threads into display tasks.

A goal of the MMI design was easy addition of new displays. Ada's packaging concept made this an especially easy goal to achieve. The goal was accomplished by having one task type that could draw any of the displays. One of these tasks is allocated for every active display. The control tasks in the MMI can then treat all display tasks the same. Display tasks reference packages that conceal the details of each different type of display. For example, a window is opened by the operator during execution for display of association results. A new task is allocated. It is initialized to function as this type of display. The task then calls subprograms in a specialized package to manage the display (redraw, get input, resize). All display packages have the same interface because all must handle the same events. Mouse events, keyboard events and window movement events are examples.

To add a new display is easy: a new package is created conforming to the standard interface and the task body is updated to reference the new package. The data collection tasks have the same extensible design as the display tasks.

A configurable number of display and data collection tasks are allocated at system initialization. This was initially appealing because it avoided the runtime overhead of task allocation and deallocation. It turned out to be necessary because the runtime system does not do garbage collection on deallocated tasks. Dynamic allocation would eventually cause the system to run out of memory and fail.

Intertask Communication

The MMI design required that all display and data collection tasks be capable of exchanging messages (for the remainder of this section display tasks and data collection tasks are collectively referred to as MMI tasks). The nature of the MMI tasks complicated this requirement. Some were transient and others were permanent, some were single instance and some were multiple. An additional requirement to support three modes of communication complicated matters. It was necessary to support broadcast, multicast, and point-to-point modes. A broadcast message is sent to all MMI tasks. A multicast message is sent to all instances of a particular type of MMI task. A point-to-point message is delivered to a particular instance of a particular type of MMI task.

Ada's *rendezvous paradigm is insufficient for this* application. First, it provides no mechanism for group communication. A rendezvous is by definition point-to-point. Second, it introduces the possibility of deadlock occuring when MMI tasks are exchanging messages. To randomly allow any module to rendezvous with any other module invites deadlock in situations such as two modules calling each other at the same time. One solution to this problem is to place restrictions on which MMI tasks can call others. A design that seeks to allow any MMI tasks to communicate precludes this alternative. Two other alternatives were considered.

The first alternative used an agent task to make the message delivery (See Figure 4). An MMI task procured a message agent and gave the message to the agent for delivery. This alternative had the advantage of being easy to understand and implement. Its disadvantages were that three rendezvous would have to occur to deliver a single message and that extra overhead would be incurred for the agent allocator task and the agent tasks.

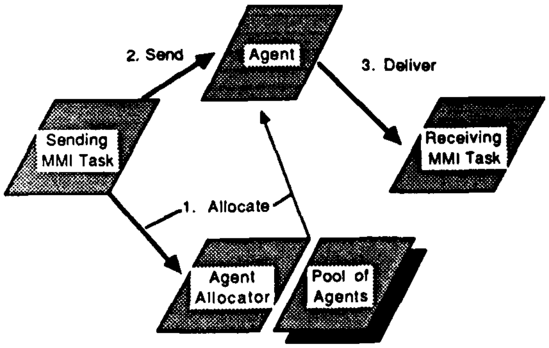


Figure 4: Use Of Agent Tasks For Message Delivery

The second alternative used shared mailboxes and a mail delivery task (See Figure 5). Each MMI task had an in-box and an out-box. An MMI task read messages from its in-box and placed messages in its out-box. The delivery task periodically visited the mailboxes and transferred messages from out-boxes to in-boxes. The advantage of this method is that it requires no rendezvous to transfer a message. It has several disadvantages. First, there is a time lag between the time that a message is placed in the out-box and the time it is received by the re.:ipient. This time lag may have a large variance depending on the number of tasks, the number of messages, and how often the delivery task executes. Second, MMI tasks have to handle encountering a full out-box when trying to send. This is especially troublesome for the delivery task attempting to deliver broadcast and multicast messages; sometimes several of the recipients have full in-boxes and the delivery task has to keep track of which recipients have received the message and which have not. Third, it requires the delivery task and the MMI tasks to poll the mailboxes.

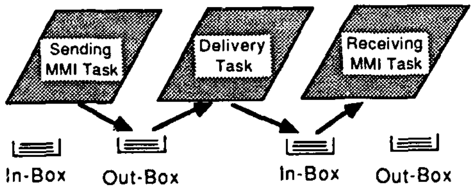


Figure 5: Use Of Mailboxes For Messaging

Both methods were tried and both met the requirements. However, the agent task method was superior to the shared mailbox method because it ensured a form of flow control not available with the shared mailbox method. Busy senders did not have to worry about filling up their mailbox; the inherent nature of the rendezvous assured that all messages were delivered.

OOD Used In The Simulator

The simulator is required to produce emitter reports for testing in absence of a real signal sorter. It provides a *generalized radar simulation that can be configured to* run different scenarios. A scenario consists of a set of moving, multiple-emitter platforms and a set of moving, multiple-receiver collection platforms. The characteristics of the emitters and receivers may be specified along with their movement patterns.

The simulator operates as follows: when it is started, each emitter and each receiver operates autonomously according to the scenario description. As receivers scan frequency bands, they detect emitters currently transmitting in that band. The receiver informs the signal processor of the emitters that were seen, their parameters, and the locations of the emitter and receiver. The signal processor then generates emitter reports from the information.

An event-driven simulation was chosen. In this type of simulation initial events are placed into an event queue in time-order of execution. When an event is at the head of the queue the event is dequeued and executed. Each event that is executed will, in general, alter the state of the simulation and generate further events to be executed. In this manner, the simulator runs indefinitely or until a preset stop condition is encountered.

The simulator lent itself particularly well to Object-Oriented Design. Figure 6 depicts the design, showing the objects with boxes and the dependencies with arrows. The event manager controls execution of the simulation. When told to start by the MMI, it tells each of the receiver objects and each of the emitter objects to seed the event queue. This will cause each receiver and

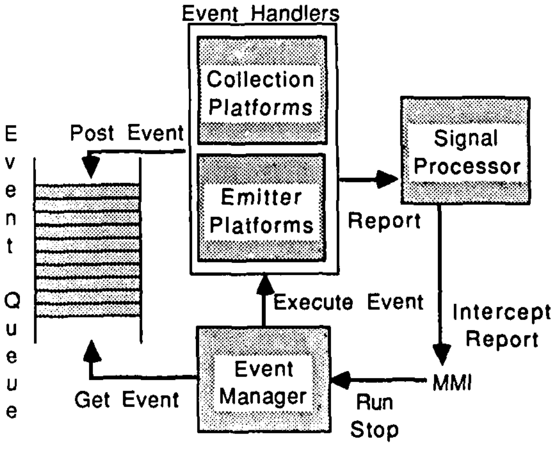


Figure 6: VHSIC TAM Simulator Object Design

emitter object to post their initial events to the event queue. When this is done, the event manager will get the next event from the event queue. It will then alert the appropriate receiver or emitter object of the event and that object will execute the event. As receivers detect emitters they inform the signal processor. The signal processor will then generate emitter reports.

Intersystem Communication

The VME bus extender provides the high bandwidth communication path between the workstation and the embedded computer. Installation of the extender was not simple; one hardware problem quickly became apparent. The bus access protocol used by the bus extender was incompatible with those used by the workstation and the embedded computer CPUs. This required a hardware modification to the bus extender.

Designing the communication across the bus extender was not straightforward. Communication needed to be bidirectional. One way to effect this would have been to allow either system to interrupt the other when there was data available in the embedded computer's shared memory[1]. Unfortunately, it was discovered that the workstation did not support generation of VME bus interrupts from software. This precluded peer communication. In the current design, the workstation is a slave to the embedded computer. More specifically, the embedded computer has to ask the workstation for reports and has to tell the workstation when results are available.

The Bus Managers

To control the bus extender on the workstation a UNIX device driver had to be written in C. This driver allowed the bus to look like a file that could be opened, closed, read, and written.

The question of how to access UNIX devices from an Ada environment is one that has subsequently been encountered many times. The problem arises because of the implementation of Ada on the workstation: an Ada program exists as a single UNIX process. Whenever any part of the program makes a blocking call on a device the entire process may get blocked. This means that a task dedicated to handling a device may cause all other tasks running in the process to also be suspended when it executes a blocking call. Ideally, only the calling task should block while all other tasks continue executing.

Two solutions were considered. In the first, a second process (Ada program) handles the device. This process blocks on the device. The two processes communicate asynchronously using UNIX signals and UNIX interprocess communication. The disadvantage of this method is that it requires interfacing UNIX signals to the Ada program.

The second solution considered does non-blocking reads and writes to the device. The advantage of this method is that it is simple to implement, requiring only a small amount of code to configure the device and to handle retries when the call fails (normally, this is where it would block). The disadvantage of this method is that it requires device polling. The second solution is used and its performance is acceptable.

A Fight With Strong Typing To Receive Messages

Many different messages are exchanged between the workstation and the embedded computer. The different kinds will, in general, be of different sizes. Ada's strong typing causes a problem when trying to read a message from the device. The message type cannot be specified because it is unknown--any type could be received next. Two solutions were considered.

The first uses a single type for all kinds of messages. It is a large variant record with many discriminants. This is not appealing because of the way that the compiler treated variant records. The code generated for the receive always specified the maximum size of the message. A real time system cannot afford to always read and write the size of the largest possible message.

The second solution avoids the type checking. A type field precedes every message and the bus managers simply read and write bytes. When a buffer of bytes is read, an access type is cast to point to the message type. Knowing the message type then allows the correct access type to be cast to point to the message. The main disadvantage of this alternative is that it avoids the strong type checking. This problem has reappeared on other projects, especially those involving networking.

[1] Bus loading would occur with three CPUs trying to access the bus. This would significantly degrade performance. To avoid this degradation, the busses remain electrically isolated until opened for data transfer.

The latter alternative has always been chosen. This is one situation where strong type checking is more of a hindrance than a help.

The Threat Association Subsystem

The threat association subsystem resides on the embedded computer. Two processors comprise this subsystem. The application processor is dedicated to running the threat association software, the AMM controller is dedicated to controlling the VHSIC hardware. The software on the application processor is a combination of Ada and 68020 assembler. The assembler is written using package MACHINE_CODE as defined in the Ada Language Reference Manual [4].

The Ada run time environment used on the application processor is an engineering release of the Verdix 68020 run time environment[1].

The interface to the AMM uses a combination of shared memory and interrupts. The AMM manager controls this interface. To perform a search, the AMM manager sets fields in the shared memory and interrupts the AMM. Upon receipt of the interrupt, the AMM firmware copies the data from shared memory into the WAMS. The VHSIC hardware performs the search and places the results into shared memory. The AMM then interrupts the AMM manager to signal the completion of the search.

This simple interface was difficult to implement with Ada. Many problems were encountered, most can be attributed to the immaturity of the runtime environment. The problems are in four categories: dynamic memory management, debugging, timer support, and interrupt support.

Dynamic Memory Management

There is no garbage collection to handle buffer deallocation. It is easy to allocate a buffer, but once allocated it cannot be freed. The VHSIC TAM solution uses static allocation; at the start of execution all buffers are preallocated. Static allocation provides continuous system operation.

Debugging

There is no embedded system debugger. This makes it extremely difficult to identify programming errors; the only way to debug is to use print statements. This method also has problems. If an interrupt occurs while doing terminal I/O, the program fails.

Timer Support

There is no software timing support based on a real hardware time˞ Timing, such as that required by an Ada "delay" statement or the Calendar package, is implemented in software and does not correspond to real time. For example, a delay statement of 10 seconds is executed and appears to return immediately. Fortunately, accurate timing is not as critical for this application as throughput. This situation would not have been acceptable for applications such as avionics which require precise timing.

Interrupt Support

Perhaps the most notable implementation problem is that the runtime system provides no support for Ada interrupts. Ada interrupts are defined to be handled as task entries, however, the compiler does not implement this part of Ada. Three solutions were tried. The first solution branches to an Ada interrupt handling procedure which then calls a task entry. This solution fails because it causes stack corruption. The second solution also branches to an interrupt handling routine, however, the routine sets a flag rather than calling a task entry. This requires the AMM manager to poll the flag to catch the search completion. This approach also causes stack corruption.

The conclusion was that interrupts could only be handled when expected. At this point the design really became one which utilizes synchronous remote procedure calls instead of asynchronous interrupts. The third solution is the same as the second except that this solution requires that the program execute a polling loop immediately after requesting a search. It does not exit the loop until the flag is set. When the *Done* interrupt is received from the AMM, the AMM manager exits the loop and retrieves the search results. This is a very undesirable solution, however, it is the only one that can be made to work predictably. Essentially, parallelism on the embedded computer could not be exploited.

[1] An engineering version is not even a Beta version of a product. As such, it cannot be expected to support many of the features needed by an actual product. This particular version was supplied to us more than a year before the actual product release.

PROJECT IMPLEMENTATION

The system was implemented using a top-down incremental development approach. This approach is especially well supported by Ada's concept of packages that have separately compilable specifications and bodies. First, a "skeleton" of the system was built. This skeleton was an executable framework of the Ada module specifications, with stubs for the Ada bodies. Interfaces to the hardware were also filled in with software stubs. The skeleton allowed expedient implementation and testing of the control paths for the entire system. As each module was completed, it was integrated into the system in place of its stub, and the system was retested. Separately compilable bodies made integration easy because only the latest versions of the bodies had to be recompiled. After all software was integrated on the workstation, one of the subsystems was moved to the embedded computer. The workstation-embedded computer interface, and the embedded computer-AMM interface were installed and tested. The final system testing and tuning was then performed. The building of the system proceeded smoothly using the skeleton and incremental integration.

Ada development environments provide several useful tools. The VADS development environment provides integrated tools for building software, cross-compiling, software configuration management, symbolic debugging, code disassembly, and pretty-printing. The most useful tool was the "make" tool used for building programs. This tool automatically kept track of inter-file dependencies. This relieves the programmer from the tremendous burden of manually tracking impacts of changes to the code. One area in which tools are lacking is software version maintenance. We used the standard UNIX tool, SCCS, to keep a history of software versions.

OBSERVATIONS AND CONCLUSIONS

VHSIC TAM has been a big success for the customer and for ESL. The system met its requirements and much has been learned about the use of Ada in embedded systems. The following list contains some observations and conclusions.

- The combination of Functional Decomposition and Object-Oriented Design methods proved to be a good way of approaching the design. They are complementary methods.

- The software engineering principles utilized by OOD in combination with Ada allows rendition of a design that is easily distributed, easily maintained, and easily extended. The current VHSIC TAM architecture is being further distributed across more CPUs in the follow-on project.

- Ada has no paradigm for distributing Ada across multiple processors. This has impact on many embedded systems that utilize multiple processors. The same old approach of building separate Ada programs for each processor must be taken until a paradigm is defined.

- The use of Ada has promoted some reuse. VHSIC TAM used some of the packages found in the public domain, for example, linked lists and variable length strings. Some generally useful packages developed to support VHSIC TAM such as thread of execution tracing and error-logging have been reused. ESL is looking at ways to make more of the MMI reusable. Ada's concept of generic packages was a big factor in achieving reuse of the software.

- Even validated compilers still have problems with generics. We ended up just using the generic package as a code template to create multiple packages because of incorrect code generation.

- Ada's rendezvous paradigm is insufficient for many types of communication. Distributed Ada modules need better support for asynchronous communication and for group communication (broadcast and multicast).

- The choice of runtime environment on workstations can greatly affect the system architecture. The device handling would have been designed differently had the implementation not required that all tasks be contained within a single process.

- Good support for chapter 14 of the Ada LRM was very important to the completion of the project. The interface to C, machine code insertions, and unchecked conversions were especially important.

- The dependency checking and bookkeeping required by the compiler and make tools cause compilation to be slow compared to other high-level languages and require large amounts of disk space.

- It would be nice to have some method of conditional compilation. We had to create a pre-processor for this function. It is particularly important for placing assertions and tracing calls into the early stages of the code.

- Ada is frustrating to try to compile, but satisfying to run. The strict type checking shifts much of the developers' work from debugging to compiling. We have found that once we get a program to compile and load that often it ran the first time. We cannot say this for other languages.

- Ada development and runtime environments are still immature (especially for embedded systems). Bugs still exist in much of the software. A bug in a compiler can greatly impact the development schedule. Timely completion of VHSIC TAM would have been much more difficult had we not had high-quality, responsive support from Verdix when problems were encountered. Projects choosing Ada environments, especially ones for embedded applications, should carefully consider the quality of support services provided by the vendor.

Acknowledgements

We extend a special thanks to Kristine Folker who provided Ada expertise when we had little, to Mike Bautista for his excellent program management, to Max Frederick for his database work, to Doug Howsen for his support, to Verdix for allowing us to try early releases of products, and to the many people who gave their time to review our work.

References

1. J.Voelker, "Ada: From Promise To Practice?", IEEE Spectrum, Vol 24(4), April, 1987.

2. G. Booch, "Object-Oriented Development", IEEE Transactions on Software Engineering, Vol SE-12(2), February, 1986.

3. G. Booch, *Software Engineering With Ada*, ISBN 0-8053-0600-5, The Benjamin/Cummings Publishing Company, Menlo Park, CA, 1983.

4. *MILITARY STANDARD Ada Programming Language* (ANSI/MIL-STD-1815A-1983).

About the Authors

Robert J. Abel received the B.S. degree in computational physics from Carnegie-Mellon University in 1981, and the M.S. degree in Computer Engineering from the University of Southern California in 1985. He is currently pursuing further graduate work at Stanford University. He is a staff engineer and manager of the Embedded Computing Department of the Advanced Technology Laboratory at ESL, Incorporated, Sunnyvale, California. His technical interests are in the areas of distributed systems, computer architecture, pattern recognition, and signal processing.

Cindy C. Dare received the B. A. degree in computer science from the University of California at Berkeley in 1982. She is currently an engineer in the Embedded Computing Department of the Advanced Technology Laboratory at ESL, Incorporated, Sunnyvale, California. Her technical interests include distributed computing and computer architecture.

Richard J. Prokop received the B.S. degree in Electrical Engineering from Cornell University in 1982. He is currently a senior engineer in the Embedded Computing Department of the Advanced Technology Laboratory at ESL, Incorporated, Sunnyvale, California. His technical interests are in the areas of distributed systems, user interfaces, and image processing.

William R. Snow received the B.S. degree in Electrical Engineering from Cornell University in 1982 and the M.S. degree in Electrical Engineering/Computer Engineering from Stanford University in 1983. He is currently pursuing an M.S. degree in Engineering Management at Stanford University. He is a senior engineer in the Embedded Computing Department of the Advanced Technology Laboratory at ESL, Incorporated, Sunnyvale, California. His technical interests are in the areas of distributed systems, computer architecture, and software development methodologies.

Abstraction and Terminology Standardization: Necessary Elements for Establishing Comparative Metrics for Software Development Methods.

L. Mahajan[†], M. Ginsberg[†], R. Guilfoyle[‡], R. Pirchner[‡]

[†]Teledyne Brown Engineering
788 Shrewsbury Avenue
Tinton Falls, N.J. 07724

[‡]Monmouth College
West Long Branch, N.J. 07764

ABSTRACT

The perceived need to evaluate and measure instantiations of the software process is discussed from two perspectives. One of these is the logical and practical requirements for measurement; the other is commentary based upon results of an effort by the authors to develop a catalog of software development methods for the software engineering community. The authors offer conjectures to be tested further, and based upon their experiences, offer opinions intended to stimulate further thought on the problem of measurement and evaluation.

1. INTRODUCTION

As Ada technology continues to mature, a critical question is facing software developers. Which of the myriad of software development methods (SDMs) are most appropriate for use with an Ada project? The answer to this question demands that the software engineering community determine how to measure and evaluate methods. As part of a recent project to develop a catalog of software development methods [Maha87], the authors considered what is required to describe and contrast SDMs. An analysis of the problem led to the following assertions:

1. *There is a desire or need to compare software development methods*. The need for some type of comparison has been expressed both by those within the software engineering community and by software clients. In 1980, B. Curtis wrote:

   "The magnitude of costs involved in software development and maintenance magnify the need for a scientific foundation to support programming standards and management decisions. The argument for employing a particular software technique is more convincing if backed by experiments demonstrating its benefit. Rigorous scientific procedures must be applied to studying the development of software systems if we are to transform programming into an engineering discipline." [Curt80]

   If one accepts the assertion that there is such a need, then the statements which follow represent a chain of logical consequences.

2. *Comparisons of methods must be based upon the identification of a suitable set of similar or common attributes*. Any type of comparison requires the establishment of some suitable set of attributes upon which to base the comparison. Consider, for example, an attempt to compare method ABC with method XYZ. Method ABC provides an approach to formulating system specifications, while method XYZ provides a framework in which to do detailed design. Since the methods do not address the same activities within the development process, it would be ludicrous to consider a comparison of the two methods with the intent of providing a user with information as to which method was "better". On the other hand, determining what modes of communication are used by the respective methods may enable a user to determine whether these two methods could be easily used in conjunction with one another.

3. *The common attributes of SDMs must be recognizable and separable*. The authors assert that it is necessary to establish suitable attributes upon which to formulate comparisons of SDMs. Additionally, it must be possible to "separate" and "uncouple" these attributes if they are to be analyzed and measured

independently from one another. For example, would it not be useful to single out which attributes of an SDM make the method well-suited for use in a project whose implementation language is Ada, or whose application domain is real-time systems?

4. *A measure must be established for each of the identified attributes*. Again in [Curt80], B. Curtis states that application of scientific procedures requires "the development of measurement techniques and the determination of cause-effect relationships." It is not sufficient to base a comparison on some vague notion of "goodness", or on some loosely defined feeling of satisfaction expressed by a user of an SDM. A suitable range of values must be chosen which correspond to the scope of the attribute, thereby providing a measurable basis for the attribute. These values need not be numeric, and the basis of comparison can be other than statistical.

5. *The existence of a measure is independent of the existence of a practical way to determine the correct values associated with an SDM.* The establishment of a suitable measure represents only a first step. The pragmatic aspect of determining correct values must be addressed. An SDM does not exist in isolation. There are users of the SDM, and the capability of such users to correctly interface with an SDM must itself be measured. Developers work within organizations, and organizational issues may affect the successful use of the method. Finally, a particular instance of use of an SDM is associated with a particular application problem, and this too can influence the results. Thus, establishing proper control over all these variables is necessary before measured values can be correctly correlated with the SDM itself.

6. *The existence of a practical way to determine values to measure a given attribute is itself independent of the interpretation of the results*. Given that a practical way has been determined to measure SDMs, care must be taken in the interpretation of results. Consider a simplistic measure, the number of successful projects for which a particular method has been used. Knowing that one method has been used successfully many more times than a second method establishes that there is a "difference". To determine causality requires further study of why this difference exists.

7. *Measurement data alone may not be sufficient for establishing causal relationships*. The fact that measurements can be taken, and that reasonable interpretations can be assigned to the results, may still not be sufficient for establishing some type of causal relationship for individual SDMs. The establishment of such a causal relationship may depend upon the results of experiments in which two or more SDMs are used for the development of the same large software project under the same circumstances. To ensure the same circumstances might require multiple developments which use comparable people, working in a comparable environment, with comparable management, and at the same point in time. The question then must be asked, "Is such an experiment feasible?"

One goal of software engineering should be to obtain correct interpretations of measures defined for SDMs. Though work has been done toward achieving this goal, it is the authors' contention that this goal has not yet been reached. In fact, based on their experience in developing a catalog of software methods, the authors contend that the prerequisites for progressing towards the goal are not yet in place.

In the process of developing a catalog, the authors established what they believe to be a useful and practical standard for contrasting SDMs by means of a set of descriptive attributes. The resulting descriptions of individual methods did differ because the methods themselves differ. Nevertheless, by choosing to use a common style and framework for the descriptions, a basis for comparing methods was established. Additionally, conclusions drawn were restricted to statements about what a method does rather than making assertions about the quality of resulting software produced using the SDM.

In the effort to produce the catalog, the authors addressed assertions 2 through 6 above. Their approach centered around adopting a standard terminology and developing abstractions and models. It was hoped that ways could be found to describe attributes of SDMs by abstracting their "essence" and postulating models exhibiting how the abstractions fit together. The authors concluded that the lack of agreement on the use of terms within the software engineering community suggests that there may well be a *new software crisis* resulting from this lack of uniformity of expression in connection with the software process. The authors propose that some of the suggestions which addressed the previous crisis are applicable to the new one: in particular, the need to standardize and foster uniformity for the bulk of development work. A note of clarification here; the previous statement is *not* a plea for use of one method; rather, it is a plea for use of standard terminology.

2. BACKGROUND

The goal in producing the catalog of software development methods was to provide interested members of the software engineering community with information about ways in which the software process is carried out. One of the means for reaching this goal was the establishment of characteristics which tell researchers about methods when:

- The acquisition or enhancement of one or more SDMs is being considered.
- Work or a proposal incorporating the use of SDMs is being evaluated.
- There is a need to describe one's own SDM to others.

After discussing techniques for comparing methods, the authors concluded that a procedure based on descriptive characteristics is the appropriate level at which to begin. In order to facilitate such a comparison, the authors decided that a uniform framework should be used to describe individual methods, and that further information should be conveyed by providing tables in which specific characteristics of methods are contrasted. Examples of types of tables included in the completed catalog are:

- Activities within the software process addressed by the method.
- The suitability of the method for specific application areas.
- Modes of communication and representation used by the method.
- Verification support provided by the method.
- Project management support provided by the method.
- Training assistance available for the method.
- Estimates of time needed to learn the method.

The catalog was developed for the US Army CECOM installation located at Ft. Monmouth, New Jersey. Information for the catalog was gathered by surveying both method developers and users of methods in the software engineering community. Survey questionnaires were distributed to approximately 150 developers, and 76 responses were returned. Of the approximately 160 user survey questionnaires which were distributed, 39 were returned. The resulting catalog contained descriptions of 47 methods.

In order to achieve their goal, the authors found that four major problems had to be addressed. These involved: contending with variations in terminology, establishing key characteristics upon which to base the descriptions of individual SDMS, formulating developer and user survey instruments, and evaluating the survey responses. An implied follow-up to the authors' work is the possibility of an enhanced catalog in which SDM products are rated according to effectiveness, utility, economy, or other measures of merit.

2.1 Contending with Terminology

What is a software development method? Is a single computer-based development support tool a method? Is a collection of such tools a method or an environment? What does it mean to say that a method fosters reusability, that use of a method is cost-effective, or that a method supports maintainability? How does one interpret terms associated with methods such as understandability, cohesiveness, applicability, and transferability? The authors concluded that addressing such questions required an agreement among themselves relative to the usage of terminology, even when that implied a dramatic change from previous practice and "understandings". This leads to the following conjecture:

> Effective intra-team communication requires a conscious agreement by development team members to use terms in a consistent manner.

How should a basis for agreement on the use of terms be established? One looks at existing efforts toward standard terminology. The authors referred to existing glossaries, such as those appearing in [IEEE83] and [Free82], and the efforts of others in the area [IWSP85]. After analyzing such works, the authors still felt the need for additional clarification. In particular, it appeared to be essential to distinguish between tools, methods, environments, and approaches. The authors' resolution of this problem is discussed in detail in section 3 of this paper.

Two techniques of analysis were developed for clarifying meanings: an abstracting technique and a modeling technique. The first operates as follows. Collect the terms which appear to be special cases of a more general concept. Next, agree upon a term or phrase to represent that concept. Examples where use was made of this technique appear in section 4 below.

The modeling technique involves the use of some representation, other than English narrative, for the subject under study. A diagram is often the form that might be used. The diagram is modified until agreement is reached on what the diagram means in the context of the software process. Ultimately, the representation developed in this manner becomes the "definition" of the concept, to be referred to when there is a need to test correctness of an assertion about the concept.

2.2 Establishing Key Concepts for SDMs

The next step in this aggregation of information was to determine a collection of key concepts - phrases which describe characteristic of SDMs and which are of practical value. The technique used by the authors to converge upon a small list of items is described in the following paragraphs.

An initial set of 125 concepts was identified based upon terms used primarily in the two Methodman studies, [Free82] and [McDo86], with some further input from other methodology research sources. This set, based upon the considerable previous efforts of others, provided a foundation for further analysis and refinement.

The candidate concepts were first rated as to their identifiability or recognizability by using a polling technique. The word "identifiability" is used to mean the capability of establishing that a given method has the given characteristic. For example, establishing the economic benefit of an SDM would be very useful, but the ability to actually identify or recognize this characteristic is anything but easy. Accordingly, each of the authors individually attempted to rate the characteristics. Then a group consensus was sought, though in a few cases the authors were forced to accept the fact that a clear consensus was not reachable.

Following this, a similar approach was used to rate the concepts according to their importance. Concepts which were considered very important, but difficult to recognize, were analyzed further by looking for inference chains which associate these important concepts with other, more easily recognized concepts. These refinements led to the selection of a set of key concepts which would form the basis for the questions in the survey instruments, and which would be used to describe SDMs in the catalog.

It was evident in this process that various studies were not consistent in the meaning assigned to such characteristics. Again it was necessary that at least among themselves, the authors would need to reach agreement on meanings. Conveying these agreed-upon meanings to others proved to be more of a problem than anticipated. This is discussed in section 2.3.

One of the lessons learned from this analysis was that certain terms deemed to be highly important were not identifiable. This seems to be due to several causes. First, the concept may represent an abstraction with more easily identified composite characteristics. An example of such a concept might be "changeability". Second, the word used may represent a "conceptual hook", a concept that came into being before any well-defined subcharacteristics had been associated with it. (The idea of a conceptual hook is discussed further in section 4.) A possible example of this latter concept is "reusability". Finally, the characteristic may be of a nature that would require some form of experimentation to demonstrate its existence. "Reliability" is an example of such a concept.

As a result of the process of determining key characteristics, the authors selected the following to describe methods:

- Activities covered by the method;
- Extent of usage;
- Appropriate application areas;
- Ability to incorporate requirements of the target system;
- Support of communication during development;
- Specification and support of client involvement;
- Support of changeability;
- Support of project management;
- Available automated tools supporting the

method;
- Available training in the use of the method;
- Acquisition factors.

Further, it was agreed to attempt to identify how individual SDMs provide assistance for producing software which can be characterized as being maintainable, portable, testable, reliable, and reusable. When discussing such characteristics, which apply more to the qualities of the resulting software than to the method, the authors chose to focus on the intent of the method to impart these qualities rather than to evaluate the effectiveness of that intent.

Finally, for purposes of establishing a uniform format for describing methods, the characteristics selected were grouped into the following major categories: technical aspects, means of expression and communication, applicability and coverage, usage of the method, project management support, training, and acquisition factors. Many of the characteristics are relevant to several of the categories, necessitating some redundancy in the presentation of information.

It may be observed that these characteristics are similar to those derived in previous studies. Where this analysis does differ from these previous studies is in the emphasis placed on the method itself as opposed to the software resulting from use of the method. Thus, in the catalog some important software qualities were not emphasized due to the lack of measurable criteria. What is provided is a uniform descriptive framework for reporting information on a wide range of methods. In the authors' opinion, it is this framework which provides a basis through which the reader can contrast methods.

2.3 Survey Development and Evaluation

How should one develop questions intended to reveal whether or not a given characteristic is present in a method? Some of the ground rules adopted by the authors for the questions to be asked were:

- Avoid the use of buzzwords, jargon and terms whose meaning is overloaded.
- When posing a question, be objective; do not introduce bias by limiting the possible answers.
- Address the need to motivate users to respond to the survey.

One premise used by the authors was that, for survey purposes, important but hard-to-identify concepts must be preserved in essence, but not in form. There were several reasons for doing so. For one, abstract goals such as "maintainability" are what motivate people to develop or choose a method. They hope that achieving such goals will contribute to a better software product. The spirit of the concept must be kept in mind because of its importance. In spite of this, however, the very words representing the goals are often used in a *symbolic* sense to impart qualities to a method or to the software that have no real basis in fact. Thus, terms which are indeed important can degenerate through usage into buzzwords and jargon.

To convey the precise meanings intended by the authors would have necessitated a presentation of extensive definitions or textual explanations. These definitions could be subject to disagreement on the part of the readers, hindering the effort to gather information. Instead, the authors elected to look for the evidence which would imply the existence of the concept being analyzed. More specific and recognizable concepts were assembled into groups, with the idea that if such concepts were shown to be present in a method, then the less recognizable concept could be inferred to be present also.

This approach to conveying meaning in the survey allowed the authors to construct a querying technique that was based upon more objective and agreed-upon words, thus avoiding as much jargon as possible. The overloading of words was avoided as well as the presentation of different words which were judged to be synonymous. The segmentation of concepts among several different questions allowed the authors to re-synthesize the abstraction from the responses to the set of component questions during data reduction and analysis.

After striving to meet the above requirements, the authors found that there still were problems to be addressed. One way to get the respondent to answer a survey is to make it simple to answer. Typically this leads to multiple-choice and check-off type questions. Often, there is a difficult trade-off between asking a complex question versus the desire to make it easy for the respondent to answer. There is a significant difficulty in that easy-answer questions tend to force things into categories that could reflect the authors' own pre-conceived notions of what the answers should be.

Another difficulty is that respondents may want to appear to be above average, better informed or more knowledgeable; consequently, they may not be candid. People may not even realize that their responses are biased. In particular, it is natural for vendors of methods to want to present their products in a favorable light. For example, how is one to evaluate the often encountered claim from vendors that their methods are suitable for almost every type of application? Because of above problem, the authors recognized the futility of asking the question, "Can your method be used to support development of an Ada project?"

After reviewing the answers provided by survey respondents, the authors judged that there still is a lack of uniformity within the development community relative to terminology. The responses showed a wide range of interpretation given to such terms as formal specification language, semantic analysis, and prototyping. In fact, there even was a wide interpretation given to such English phrases as, "activities supported by the method", "the capability of the method to address timing constraints", and "automated support provided by a method". Given the difficulty of producing unambiguous English, one must have precisely defined technical terms.

Accordingly, the authors suggest that the conjecture of section 2.1 should be expanded to:

> Effective *software engineering community* communication requires a conscious agreement by community members to use terms in a consistent manner.

3. STRIVING FOR STANDARD TERMINOLOGY

One of the first tasks which the authors set for themselves was to clarify the concepts of approach, method, environment, and tool. In undertaking this task the authors had a mutual desire to be consistent both in their own work and with the general usage of these terms within the industry. Initially, it was thought that the catalog itself should be divided into sections corresponding to these classifications. Based on ideas found in [IWSP85], the authors arrived at the definitions presented below.

> A *software development method* is a definite, established, logical, or systematic plan. The steps and purposes have been thought out beforehand in detail. A method must guide the user to a predictable result given an appropriate set of starting conditions.
>
> An *approach* is a way of beginning or managing an effort; a way of analyzing, planning or directing a project; a way of conducting operations. An approach suggests ways to identify goals initially and/or suggests, at an abstract level, ways to proceed toward goals.
>
> A *tool* is anything used to do specialized work or to obtain a specific result; there is a unity of purpose. A tool is essentially automatic: the user supplies input data or changes, but the tool produces the associated work product.
>
> An *environment* is an integrated collection of tools supporting an approach. The components of an environment are designed to reduce the effort required to carry out the software development process whether they are used individually or in combination.

It should be noted that early in the project, the authors decided that the often-used term "methodology" is overloaded and has achieved a buzzword status. Accordingly, the authors have chosen to follow the leadership of others [IWSP85] by restricting the meaning of this term to *the study of methods*.

Despite the authors' wish to make clear distinctions, attempts to classify objects according to the above definitions quickly led to difficulty. Both tools and approaches exist over wide spectra. Within these spectra, making a clear distinction between a sophisticated tool and a method on the one hand, or between a method and a prescriptive approach on the other, is difficult. Thus, the authors' use of the term *method* may include objects that could be termed tools or approaches.

By way of examples, consider the following. An integrated set of tools with a prescriptive user's manual might be offered as a software development environment. In such a case, an implied method or approach exists by virtue of the fact that there is only one way to use the set of tools in order to arrive at a final software product. On the other hand, consider a way of doing things which prescribes the order in which activities occur and how these activities are to be managed, but is not limited to one specific way of accomplishing individual activities. This might be offered as an approach. Nevertheless, the instantiation of such an approach implies a definite systematic plan for developing software.

Consider the problem of classifying a given object as either a tool or a method. The results of the survey showed that many methods are embodied in an automated tool. In some cases, the method may be extracted from the tool without losing the essence of the method. In other cases, the tool and the method are so closely coupled that the method cannot stand alone.

The way in which a method develops has an influence on its relationship to a tool. Methods which evolved without incorporating software tools may later receive support from a number of different tool vendors. Other developers treat automated support as a necessary component of their method, in which case the method and tool are likely to be synonymous.

Tools may also extend the scope of a method by addressing concepts beyond those originally considered by the method. In this case, it is difficult to distinguish whether the method has been redefined to encompass these additional concerns, or continues as but one part of a more elaborate scheme. In such a case, it can be difficult to determine whether the method incorporated the tool, or the tool subsumed the method.

This difficulty becomes even more complicated as a method which can exist independently disassociates itself from the tool in which it was first "incarnated". On one hand, the method developer may propose new aspects for the method that are not supported by the tool. On the other hand, the toolmaker may have his own ideas about the way development should be done, diverging from the developer's original ideas. A relationship still exists, but the ease with which the components of this relationship may be distinguished varies considerably.

The existence of these complex relationships reveals why it is difficult to clearly categorize methods and tools. Though there are objects which are clearly tools, and other objects which are clearly methods, there is a broad spectrum over which these two types of objects merge.

Similar complex relationships also exist among methods, approaches, and environments. It is possible to formulate carefully chosen definitions which delineate these objects, but whether there exists a well-defined boundary separating them is not well understood. The assumption that instances of the above objects are clearly classifiable may be untenable. On the other hand, the authors question the importance of making such distinctions. The issue that needs to be addressed is to what extent assistance is provided by such objects for the software development process.

4. USING ABSTRACTION TO CLARIFY IDEAS

There is a need for software engineers to communicate technical information without getting caught in language traps. Many times the exchange of technical information and ideas is hindered because the software engineering discipline has not established standardized meanings for terms. While the IEEE Glossary of 1983 includes quite a few of the terms in use, the authors did not always find a level of detail which would allow common acceptance of meaning for these terms. As a result, people involved in the study and use of methods invent their own definitions leading to even greater confusion.

This appears to be natural. Russell Abbott in his article, "Program Design by Informal English Descriptions" [Abbo83] presents the idea of a "conceptual hook" to show how people talk about something that has an acknowledged importance without exactly knowing how to actualize the concept in reality. He points out that this mode of thinking is quite different from the way data types must be specified in programs, that is, defined completely to the most detailed level.

At some point, however, it is necessary to qualify what is meant by an abstract concept, such as portability or reusability, just as it is necessary to provide a body for an Ada specification. It is at this stage that discussions become bogged down in a quagmire of terminology, jargon, and buzzwords. Thus, there is a definite challenge to progress beyond such a situation to one with a more solid foundation of ideas, upon which an increased understanding of software engineering may be built. One of the efforts to meet this challenge of terminology involves the recognition of overloading and synonymy. Overloading occurs when the same term is used for essentially different meanings, while synonymy occurs when different terms are used to represent the same basic concept. In either case, the problem of interpretation arises because the context in which the word is used is implicit, rather than explicit.

The authors' experience with the search for meaning led them to derive the procedure described in section 2.2. This procedure was based upon the acceptance of intuitive understandings of word meanings. Their premise was that consideration of both the importance of a concept or characteristic in the context of the software process, as well as its identifiability (the ability to establish its presence) would assist the authors in reaching agreement on the concept's implicit meaning.

The above procedure did not prove to be totally satisfactory for establishing a clear understanding of concepts. Further efforts to resolve uncertainties with terminology involved the use of abstraction and modeling. Abstraction is a mechanism that has been incorporated in programming languages to facilitate the design process. In fact, Ada's separation of specification from body accommodates the tendency for people to think in terms of abstractions before those abstractions are well-defined. The authors employed the abstraction mechanism as a second component in the effort to clarify terminology.

As an example of a way in which a lower-level concept can be assigned meaning, consider the following. In the realm of hardware engineering, there are a multitude of terms representing military standards for equipment. The terms "ruggedized", "human-engineered", and "climatized" seem to have little in common other than representing qualities of the equipment. They can, however, be defined as particularizations of an abstraction; and so, their relationship is seen through the abstraction. By utilizing the more abstract concept, "transportability", as the ancestor of these three terms, one might find a way to elicit better understanding as follows:

ruggedized <= transportability (vibration)
+ transportability (shock)

human-engineered <= transportability (weight)
+ transportability (size)

climatized <= transportability (geography)
+ transportability (seasons)

The above format "transportability (size)" should be read: "considering transportability in the context of size". Operating in the above manner provides reasoning advantages. For example, tradeoffs between the different contexts for transportability can be evaluated.

Abstractions can also be attained by shifting from low-level to high-level, as is common in any search for unifying concepts. For example, consider the problem of unifying the concepts of changeability, reusability, and portability. One possible unification might be the idea of polymorphism. In the notation above we have:

polymorphic (requirements) = changeability
polymorphic (applications) = reusability
polymorphic (machines) = portability

Code is portable if it can be adapted to many types of machines; code is reusable if it can be adapted to many types of applications; code is changeable if it can be adapted to many types of requirements.

The formulation above shows that if reusability is to be defined, then the characteristics of applications must be considered. In a similar vein, consider how the above mechanisms might help to put established concepts in a new light. Prototyping methods and evolutionary methods have characteristics in common; they both set up a working system prior to the development of the final system. Why get a system running "early" in the total process? An answer might be that the methods adhere to a strategy centered around intense client involvement; that is, the "client-involvement" approach. The notation above yields:

client-involvement (actual-product)
= evolutionary development

client-involvement (modeled-product)
= prototyping development

An abstraction has been formulated which may be particularized to different methods, thereby establishing the possibility that the models for these methods may have substantial similarities.

5. CONCLUSIONS

As a result of the authors' experience with preparing a catalog of software methods, several conclusions were reached. They are listed below.

1. Successful evaluation depends upon standardization of terms. So far, an appropriate level of terminology standardization related to software development methods has not been reached.
2. The characterization of software development methods must be standardized; that is, there should be substantial agreement about how to describe a method. As yet, there is no such agreement.
3. Establishing metrics on methods requires that the previously mentioned levels of uniformity be reached and that practical means for determining values of a measure be derived. Presently, there are software metrics but their correlation to the process of software development is uncertain.

The authors believe that in addition to metrics, there is a need to determine sufficient preconditions for the correct execution of an SDM, that is, an execution which will result in a successful development of an application/system. Three components of these preconditions have to do with the nature of the application to be addressed, the characteristics of the development organization, and the robustness of the method itself. In the authors' experience, SDMs do not provide adequate information to make judgements as to whether the preconditions have been, or can be, met. There is a question of "fit" here; one is trying to learn how well the problem, the method, the developer and the result all match each other. For the Ada community, questions to be asked are: "What does the method expect of my organization to get an effective implementation done in Ada? To what degree will my organization have to change to successfully use the method? To what degree can the method be tailored with little negative impact?"

For the Ada community, the need for standardization has always been a necessary and accepted fact of life. Let this standardization also be applied to terminology and to the descriptions used for methods.

BIBLIOGRAPHY

[Abbo83] R. J. Abbott, "Program Design by Informal English Descriptions", *Communications of the ACM*, Vol. 26, Nov. 1983.

[Curt80] B. Curtis, "Measurement and Experimentation in Software Engineering", *Proceedings of the IEEE*, 1980, Vol. 68, No. 9.

[Free82] P. Freeman and A. I. Wasserman, "Software Development Methodologies and Ada; Ada Methodologies Concepts and Requirements, Ada Methodology Questionnaire Summary, Comparing Software Design", University of California, Irvine, CA., Nov. 1982.

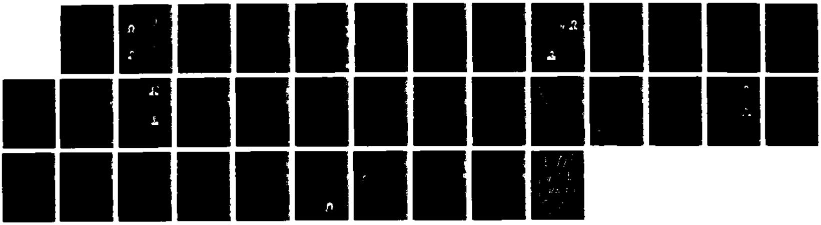
[IEEE83] IEEE Standard Glossary of Software Engineering Terminology, New York, NY, Feb. 1983.

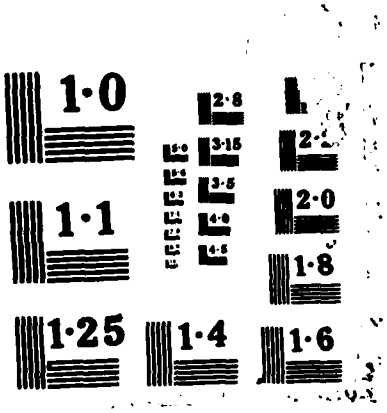
[IWSP85] SIGSOFT *Software Engineering Notes*, Special Issue - International Workshop of the Software Process and Software Environments, Vol. 11, Aug. 1986.

[Maha87] L. Mahajan, M. Ginsberg, R. Pirchner, and R. Guilfoyle, "Software Methodology Catalog", Technical Report MC87-COMM/ADP-0036, Teledyne Brown Engineering, Tinton Falls, NJ, prepared for U.S. Army CECOM, Fort Monmouth, NJ.

[McDo86] C. W. McDonald, W. Riddle, [illegible] Youngblut, "STARS Meth[illegible] Summary", SIGSOFT S[illegible] *Engineering Notes*, V[illegible]

AD-A190 936 PROCEEDINGS OF THE NATIONAL CONFERENCE ON ADA (TRADE 7/7
MARK) TECHNOLOGY (6TH) HELD ON 14-18 MARCH 1988 IN
ARLINGTON VA(U) NORFOLK STATE UNIV VA MAR 88
UNCLASSIFIED F/G 12/5 NL




1·0
2·8
3·15
3·5
4·0
4·5
5·0
5·6
6·3
7·1
8·0
2·5
2·2
2·0
1·8
1·1
1·25
1·4
1·6

AUTHORS

Laurel Mahajan is a programmer analyst at the New Jersey office of Teledyne Brown Engineering (TBE). She was task leader for the Software Methodology Catalog. She received a BA in English from the University of Chicago, an MS in Education from Northern Illinois U., and an MS in Computer Science from Monmouth College. her interests include methodology and linguistics. She is a member of ACM, SIGSOFT, and of the IEEE Computer Society.

Marilyn Ginsberg is a principal systems analyst at TBE where she participated in the development of the Software Methodology Catalog. She has also performed independent verification and validation of Government procured software, including Ada support systems. Her 18 years experience in software engineering includes the areas of software development tools, testing standards, applications development, software support, testing, and software quality assurance. She previously worked for AT&T and Concurrent Computer Corp. She received a BA in Mathematics from Rutgers U., and is currently enrolled in the graduate computer science program at Monmouth College. She is a member of the ACM, SIGSOFT, SIGADA, SIGMETRICS, and of the IEEE Computer Society.

Richard Guilfoyle is a Professor of Mathematics at Monmouth College. He has taught mathematics and computer science courses for over two decades and he consults in the areas of simulation, numerical analysis and programming languages. He received a PhD in Mathematics from Stevens Institute of Technology. He contributed to the software catalog effort while engaged as a consultant for TBE. He is a member of the ACM, SIGPLAN, SIGSIM, SIGSOFT, and of the IEEE Computer Society.

Richard Pirchner is an Associate Professor of Computer Science at Monmouth College. He received an MS in Mathematics from St. John's U. and an MS in Computer Science from Rutgers University. His areas of interest include software engineering, programming languages, database systems, and computer science education. He contributed to the software catalog effort while engaged as a consultant for TBE. He is a member of ACM, SIGADA, SIGMOD, and the IEEE Computer Society.

Lessons Learned in the Implementation Phase of a Large Ada™ Project

Carolyn E. Brophy[1], Sara Godfrey[2],
William W. Agresti[3], and Victor R. Basili[1]

1 Department of Computer Science
University of Maryland
College Park, MD. 20742

2 Goddard Space Flight Center
Code 552.1
Greenbelt, MD. 20771

3 Computer Sciences Corporation
System Sciences Division
8728 Colesville Road
Silver Spring, MD. 20910

Abstract

We need to understand the effects that introducing Ada has on the software development environment. This paper is about the lessons learned from an ongoing Ada project in the Flight Dynamics division of the NASA Goddard Space Flight Center. It is part of a series of lessons learned documents being written for each development phase.

FORTRAN is the usual development language is this environment. This project is one of the first to use Ada in this environment. The experiment consists of the development of two spacecraft dynamics simulators. One is done in FORTRAN with the usual development techniques, and the other is done with Ada. The Ada simulator is 135,000 lines of code (LOC), and the FORTRAN simulator is 45,000 LOC.

We want to record the problems and successes which occurred during implementation. Topics which will be dealt with include (1) use of nesting vs. library units, (2) code reading, (3) unit testing, and (4) lessons learned using special Ada features.

It is important to remember that these results are derived from one specific environment; we must be very careful when extrapolating to other environments. However, we believe this is a good beginning to a better understanding of Ada use in production environments.

Ada incorporates many software development concepts; it is much more than "just another language". As such, we need to understand the effects of introducing Ada into the software development environment. This paper concentrates on the lessons learned from an ongoing Ada project in the Flight Dynamics Division of the NASA Goddard Space Flight Center (GSFC). The Ada project is sponsored by the GSFC Software Engineering Laboratory (SEL). It is part of a series of lessons learned documents being written for each development phase.

Environment

FORTRAN is the usual development language in this environment. The flight dynamics applications involve mission analysis and spacecraft orbit and attitude determination and control. Many of the software development projects are similar from mission to mission providing, for example, an attitude ground support system or an attitude dynamics simulator. This pattern of developing similar applications is important for domain expertise and for the legacy developed in this environment for code, designs, expectations and intuitions. The similarity between projects allows a high level of reuse of both design and code. Since the problems are basically familiar ones, the development methodologies which involve much iteration do not seem to be necessary. The waterfall development model is basically used here, and seems to work well in this case. Lessons learned from the initial uses of Ada do not include changing this basic methodology.

Project

The project was originally designed as a parallel study with two teams. Each would develop a spacecraft dynamics simulator, one with FORTRAN as the implementation language, and one with Ada as the implementation language. The specifications for each simulator were the same, supporting the upcoming Gamma Ray Observatory (GRO) mission. However, there are many

Ada is a trademark of the U.S. Department of Defense - Ada Joint Program Office.

Contact: Carolyn Brophy, Department. of Computer Science, University of Maryland, College Park, MD 20742, (301) 454-6154.

Support for this research provided by NASA grant NSG-5123 to the University of Maryland.

other differences between the projects which keep the study from being truly "parallel". The FORTRAN version was the production version, thus they had scheduling pressures the Ada team did not have. Without scheduling pressures, the Ada team made enhancements in their version not required by the specifications, which increased time spent on the project. This was also the first time any of these team members had done an Ada project, while the FORTRAN team was quite experienced with the use of FORTRAN. The Ada team required training in the language and development methodologies associated with Ada, while the FORTRAN team did things in the usual way [McGarry, Page et al. 83]. The Ada team also experimented with various design methodologies; this was necessary to find which ones would work better for this development environment. The FORTRAN team was working with a mature and stable environment. In switching to Ada, the legacy of reuse for design, code, intuitions and experience are gone, and will be rebuilt slowly in the new language.

The philosophies of development were quite different between the two projects. The Ada team consistently applied the ideas of data abstraction and information hiding to their design development. The FORTRAN development used structural decomposition methods.

Our goals with this project include:

(1) How is the use of Ada characterized in this environment?

(2) How should the existing development process be modified to best changeover from FORTRAN to Ada?

(3) What problems have been encountered in development? What ways have we found to deal with them?

Current Project Status

Both the FORTRAN and Ada teams started in January, 1985. The Ada team began with training in Ada, while the FORTRAN team immediately began requirements analysis. The FORTRAN team delivered its product (45K) after completing acceptance testing in June, 1987. The Ada team is scheduled to finish system testing its 135K product in February, 1988. Discussions of the product size differences and effort distributions are presented in [McGarry, Agresti 88].

The lessons learned from major phases in the Ada development are being recorded in a series of SEL reports: Ada training [Murphy, Stark 85], design [Godfrey, Brophy 87], and implementation [in preparation]. This paper presents some of the main results from the implementation (code and unit test) lessons learned.

Lessons Learned

1. Nesting vs. Library Units

1.1 The flat structure produced by using library units has advantages over a heavily nested structure.

Nesting has many effects on the resulting product. The primary advantage of nesting is that it enforces the principle of information hiding structurally, because of the Ada visibility rules. Whereas with library units, the only way to avoid violations of information hiding is through self-discipline. In addition, the dot notation tells the package where a module is located.

There are quite a few disadvantages to nesting, however. Nesting makes reuse more difficult. A second dynamics simulator in Ada is now being developed which can reuse up to 40% of the Ada project's code. But in order to reuse it, the nested code has to be unnested, since the new application only needs some of the nested units. This is often a labor intensive operation. Nesting also increases the amount of recompilation required when changes are made, since Ada assumes dependencies between even sibling nested objects/procedures, even when the dependency is not really there. This requires more parts of the system to be recompiled than is necessary when more library units are used. It is also harder to trace problems back through nested levels than it is through levels of library units. There is no easy way to tell where a unit of code was called from, when it is nested. But library units have the "with" clauses to identify the source of a piece of code. For this reason it is now believed that over use of nesting at the expense of using more library units makes maintenance harder. This is contrary to the team's earlier expectations. The team had used nesting successfully before on a 5000 lines of code training project. However, this kind of approach does not scale-up well when developing large projects.

Library units seem to have a lot of advantages. Besides fewer recompilations when changes are made, and easier unit testing, every library unit can easily be made visible to any other library unit merely by use of the "with" clause. In nested units this visibility does not exist, and a debugger becomes essential to see what is happening at the deeper levels that are not within the scope of the test driver. Library units allow smaller components, smaller files, smaller compilation units, and less duplication of code. The system is more maintainable, since it is easier to find the unit desired. Reuse with library units is also easier, since the parts of the system are smaller. Configuration control is also easier with library units since more pieces are separate (i.e., the ratio of changes to code segments modified is closer to 1). The major disadvantage seems to be that a complicated library structure develops, which can lead to errors by the developers. However, if the Ada project were to be done over now, the team would use more library units, and nest less.

Advantages and Disadvantages of
Nesting vs. Library Units

NESTING

Advantages

* information hiding
* visibility control
* type declarations in one place

Disadvantages

* enlarged code
* more recompilations
* harder to trace problems through nested levels
* can't easily tell where a unit of code called from
* type declarations in one plare means problems for reuse
* harder maintenance
* debugger required
* larger unit sizes inhibit code reading
* harder to reuse part of the system

LIBRARY UNITS

Advantages

* fewer recompilations
* easier unit testing
* smaller components
* smaller files
* smaller compilation units
* less code duplication
* easier maintenance
* "with" clauses show source of other code units used
* easier reuse
* easier configuration control

Disadvantages

* no information hiding
* complex library structure

1.2 The balance between nesting and library units is an important implementation issue, not a design issue.

The issue of whether to use library units or nested units first arises in the design phase. At least this is the case if it is assumed that the design documents reflect this aspect of implementation (i.e., the design documents indicate in some way when nesting is intended vs. when library units should be used). While it is appropriate for the design to show dependencies, these should not dictate implementation, as far as the library unit/nesting question is concerned. The team considered the decisions concerning nesting/library units to be an implementation issue.

The library units in the Ada project went down about 3 to 4 levels, while nesting went down many levels below that. Another view of the system shows the Ada project had 124 packages and 55 library units. During implementation most team members felt an appropriate balance had been reached between nesting levels and number of library units. However, in retrospect, several felt the nesting had been overdone.

1.3 It appears best to use library units at least down to the subsystem level, and nesting at lower levels where there is minimal interaction among a small number of modules.

Experiences with unit testing seem to indicate that library units should at least go down to the subsystem level. This makes testing easier. Below this level the benefits of nesting sometimes become too important to ignore. This is one heuristic which could be used to help determine when the transition from library units to nested units should occur.

An additional way to determine when the transition should occur is to examine the degree of interaction between pieces. For modules which interact heavily, library units are preferred. At the point where the interaction drops off, using nested units is preferable. Sections with nested code are easier to deal with when they are small.

1.4 In mapping design to code, caution should be used in applying too rigorous a set of rules for visibility control.

In an attempt to control visibility, two features appear to have been too rigorously applied. The first feature is nesting. The design of the Ada project seemed to suggest a particular nesting implementation. But this created many objects within objects yielding a high degree of nesting. The second way to control visibility is through the use of many "call-through"s (a procedure whose only function is to call another routine). "Call-through"s were used to group appropriate pieces together exactly as represented in the design. They can be implemented via nesting or library units. Faithfulness to the design structure was maintained this way.

The design had non-primitive objects with specific operations. These objects were implemented as packages. To put the specific operations (subprograms) into the objects (packages) the team used "call-through"s. Thus a physical piece of code was created for every object in the design. "Call-through"s are one of the reasons for the expanded code in the Ada project when compared to the FORTRAN version. It is estimated that out of the 135K LOC making up the Ada system, 22K LOC (specifications and bodies) are because of "call-through"s. While "call-through"s provide a good way to collect things into subsystems, these should be limited to only two or three levels in the future.

If the implementation were to be done over now, many of the existing "call-through"s would be eliminated. Instead of creating actual code to correspond with every object in the design, some objects in the design would remain "logical objects". No actual packages would exist; instead, a logical object would be made up of a collection of lower level objects.

2. Code Reading

Code reading is generally done with unit testing. The developer doing the code reading is not the one who developed the code. Comments are returned to the original developer. After code reading and unit testing, the unit is put under configuration control.

2.1 Code reading helps in training people to use Ada.

Besides helping to find errors, code reading has the benefit of increasing the proficiency of team members in Ada. Individuals can see new ways to handle the algorithms being encoded. Code reading also allows another person besides the original developer to understand a given part of the project. This insight should help understanding and lead to better solutions of problems in the future.

2.2 Code reading helps isolate style and logic errors.

The most common errors found in code reading with Ada were style errors. The style errors involved adding or deleting comments, format changes, and changes to debug code (not left in the final product). Other types of errors found are initialization errors, and problems with incompatibilities between design and code. This can be due to either a design error or a coding error.

Because the Ada compiler exposes many errors not exposable by a FORTRAN compiler, code reading Ada has a different flavor than code reading FORTRAN. For example, the Ada compiler exposes such errors as (1) wrong data types, (2) call sequencing errors, (3) variable errors-- either the variable is declared and never used, or it is used without being declared. So, one seasoned FORTRAN developer working on the Ada project noted that code reading is more interesting in FORTRAN, since there were more interesting errors found in code reading FORTRAN, not found in reading Ada code. In general, logic errors are hard to find in this application domain, but enough logic errors are found to make code reading worthwhile.

Some of the difficulty of code reading with Ada on this project was due to the heavy nesting and the number of "call-through" units. Code reading would have been helped by a flatter implementation. The SEPARATE facility makes it necessary to look in many places at once to follow the code. However, code reading in Ada was easier than in FORTRAN because the code was more English-like, and hence, more readable. Often the reused FORTRAN code is an older variety without the structured constructs available in later versions.

Code reading tended to miss errors that spanned multiple units. This would be expected with any implementation language as well. One example was a problem where records were skipped when they were being output. The debugger actually found the problem.

Despite the implementation language, code reading appears to be important for highly algorithmic routines.

Groups of routines that are used only to call others may be checked to make sure the design's purity is maintained.

3. Unit Testing

3.1 Unit testing was found to be harder with Ada than with FORTRAN.

The FORTRAN units are already relatively isolated; this makes unit testing easy. Only the global COMMONs need to be added to do the unit tests. On the other hand, the Ada units require a lot of "with'd in" code, and are much more interdependent. Another very different Ada project had perhaps even more interdependence between its modules than the Ada project did. That team also found the interdependence made unit testing very difficult. More interdependence exists between Ada units because there are more relations to express in Ada. There are textual inclusion (nesting), with-ing in (library units), and invocation. FORTRAN only has invocation.

3.2 The introduction of Ada as the implementation language changed the unit testing methods dramatically.

Unit testing with Ada was done very differently. Since one unit depends on many others, it is usually hard to test a unit in isolation, so this was generally not done. The Ada pieces were integrated up to the package level, and then unit testing was done. Then testing was done with groups of units that logically fit together, rather than actual unit testing. The integrated units are tested, choosing only a subset of possible paths at a time. The debugger is used to look at a specific unit, since the test drivers cannot "see" the nested ones. With Ada projects a debugger becomes essential. This is in contrast to the usual development in FORTRAN where no integration occurs at all until after unit testing.

This shows that the biggest difference between the way FORTRAN and Ada projects are done at this point in development is incremental integration. This actually represents a change in the development lifecycle of an Ada product; integration and unit testing are alternately done rather than finishing unit testing before integration.

3.3 The library unit/nesting level issue directly affects the difficulty of unit testing.

The greater the nesting level, the more difficult unit testing is, since the lower level units in the subsystem are not in the scope of the test driver. This is the primary reason a debugger becomes a required testing aid with Ada projects. For this reason, more library units and less nesting would have made testing easier. Library units have to go down to a level in the design that makes testing more feasible. With the Ada project that would have meant taking library units down to a lower level in the design, if the project were to be done over.

Two other ways to deal with the nesting during unit test were tried and were not very successful. One solution pulls an inner package out, and includes the types and "with'd in" modules the outer package used in order to execute the inner one. This is difficult to do for each unit. The other solution is to modify the specifications of the outer package so that nested packages can be "seen" by the test driver. This solution requires lots of recompilation. With more library units, there would be less recompilation, because there would be fewer changes of specifications. Again however, the best way to test was to use the debugger on unaltered code.

3.4 The importance of unit testing seems to be more related to application area than to implementation language.

Whether the implementation is in FORTRAN or Ada, does not seem as important as whether the application has lots of calculations or has lots of data manipulations. Unit testing seemed more valuable with scientific applications; perhaps because calculation errors show up when only a small amount of localized code is executed. But data manipulation errors require more of the system to be operating before it is known if errors are present.

4. Use of Ada's Special Features

4.1 Separation of specifications and bodies is quite beneficial and easy to implement.

The team entered the specifications first, whenever possible, before the rest of the code. This gave a high level view of the system early in the development. Another benefit is that this helped clarify the interfaces early. Separating the specifications and bodies also reduces the amount of recompilation required when changes are made.

4.2 Generics were fairly easy to implement and they reduce the amount of code required.

The only problems encountered were with correct compilation of the generics in some cases, due to compiler bugs in an early version of the compiler, rather than incorrect code. As Ada matures, this will not be a problem at all.

4.3 Using too many types increases coding difficulty.

The strong typing was very difficult to get used to, when one is accustomed to weakly typed languages such as FORTRAN. It was easy to create too many new types as well.

Often a brand new type was created with a strict range appropriate for one portion of the application. Then in other areas where subtypes could have been used, the range on the original type was found to be too restrictive, so another brand new type was created instead to handle the new situation. Then a whole new set of operations had to be created as well for the additional new type. Next time the team would recommend creating a more general new type, and using many different subtypes of the original type, rather than creating more new types. In this way operations can be reused and there are far fewer main types to keep track of. Designers need to spend time developing families of types that inherit properties from one another.

The strong typing presented some problems when testing units, though it prevents some kinds of errors, also. It was harder to write test drivers that could deal with all the types in the units being tested. It was also harder to do the I/O, since so many types had to be dealt with.

4.4 Tasking was difficult to code and test, however, this seems due to concurrency in general and not Ada specifically.

Tasks were used in the user interface part of the project. The user was given many options which made the interactions between the tasks of the subsystem very difficult to plan and execute correctly.

It was harder to code tasks from the design than it was to code other types of units. However, this is not really due to Ada, but rather it is the nature of concurrency problems. The language made the use of tasking easier, and encouraged the developers to use tasking more than they would have otherwise. The dynamic relationships of concurrency cannot be represented in the design (termination, rendezvous, multiple threads of control). Correctness was very difficult to assure, as is usual with these kinds of problems, and deadlock was hard to avoid. Functional testing was done, which is the usual type in this environment.

The major problem the developers had was with exceptions. These are no worse with tasking than they are with any other program unit, however.

4.5 Exception handlers have to be coded carefully.

The key problem with exceptions is deciding the best way to handle them. Errors and the sources of errors can be hard to find if the exception handlers are not coded carefully. Suppose a particular procedure will call another unit, expecting some function to be performed, and certain kinds of data to be returned. If an exception is raised and handled in the called unit, and it is non-specific for the problem raising the exception (e.g., "when others") , the caller gets control back without the required function being performed. But the exception was handled and data was returned, so the call looks successful. Yet as soon as the caller tried to use the data from the routine where the exception was raised and handled, it fails. Because of propagation, it can be very difficult to trace back the error to the original source of the problem.

Several members of the team would recommend incorporating the way exceptions are to be handled into the design, rather than leaving this until implementation. Put into the design (1) what exception would be raised, (2) where it will be handled, and (3) what should happen.

Ada Features*

	implementation ease	benefit
tasking	-	+
generics	+	++
strong typing	0	0
exception handling	0	+
nesting	+	-
separate specs/bodies	++	++

* This figure represents a subjective assessment based on team member interviews

Summary

We have learned several important things about four major areas in implementation. There are many advantages to using library units, though nesting can have its usefulness at some point below the subsystem level. Code reading helps train people in Ada, and helps to isolate style and logic errors. Unit testing was substantially changed by using Ada: the first stages of integration often began before unit testing proceeded. Some Ada features are quite powerful and should be carefully used.

It is important to remember that these results are derived from one specific environment. We must be very careful when extrapolating to other environments. There are also many questions still left to be answered. Studies of this project will continue, and other Ada projects are being started. These will help us evaluate the effects on longer term issues such as reuse and maintainability of the Ada projects. We believe this project is a good beginning to a better understanding of Ada use in production environments.

Acknowledgements

The Ada experiment is managed by Frank McGarry of NASA/GSFC. The authors would like to thank him and the Ada team for their cooperation and assistance.

References

[Agresti 85]
Agresti W., "Ada Experiment: Lessons Learned (Training/Requirements Analysis Phase)", Goddard Space Flight Center, Greenbelt, MD 20771, August 1985.

[Godfrey, Brophy 87]
SEL-87-004, "Assessing the Ada Design Process and Its Implications: A Case Study", Godfrey S., and Brophy C., Goddard Space Flight Center, Greenbelt, MD 20771, July 1987.

[McGarry, Agresti 88]
"Measuring Ada for Software Development in the Software Engineering Laboratory", Hawaii International Conference on Systems Science, January, 1988.

[McGarry, Nelson 85]
McGarry F., and Nelson R., "An Experiment with Ada – The GRO Dynamics Simulator Project Plan," Goddard Space Flight Center, Greenbelt, MD 20771, April 1985.

[McGarry, Page et al. 83]
SEL-81-205, "Recommended Approach to Software Development", McGarry F., Page J., Eslinger S., Church V., and Merwarth P., Goddard Space Flight Center, Greenbelt, MD 20771, April 1983.

[Murphy, Stark 85]
SEL-85-002, "Ada Training Evaluation and Recommendations from the Gamma Ray Observatory Ada Development Team", Murphy R., and Stark M., Goddard Space Flight Center, Greenbelt, MD 20771, October 1985.

Biographies

Carolyn E. Brophy is a graduate research assistant at the University of Maryland, College Park. Her research interests are in software engineering, and she is working with the NASA Goddard Software Engineering Laboratory. Ms. Brophy received a B.S. degree from the University of Pittsburgh in biology and pharmacy. She is a member of ACM.

Sara H. Godfrey is with Goddard Space Flight Center in Greenbelt, Maryland, where she has been working with the NASA Goddard Software Engineering Laboratory. She received a B.S. degree from the University of Maryland in mathematics. (picture missing)

William W. Agresti is with Computer Sciences Corporation in Silver Spring, Maryland. His applied research and development projects support the Software Engineering Laboratory at NASA's Goddard Space Flight Center. His research interests are in software process engineering, and he recently completed the tutorial text, *New Paradigms for Software Development*, for the IEEE Computer Society. From 1973-83 he held various faculty and administrative positions at the University of Michigan-Dearborn. He received the B.S. degree from Case Western Reserve University, the M.S. and Ph.D. from New York University.

Victor R. Basili is Professor and Chairman of the Computer Science Department at the University of Maryland, College Park, Maryland. He was involved in the design and development of several software projects, including the SIMPL family of programming languages. He is currently measuring and evaluating software development in industrial and government settings and has consulted with many agencies and organizations, including IBM, GE, CSC, GTE, MCC, AT&T, Motorola, HP, NRL, NSWC, and NASA.

He is one of the founders and principals in the Software Engineering Laboratory, a joint venture between NASA Goddard Space Flight Center, the University of Maryland and Computer Sciences Corporation, established in 1976. He has been working on the development of quantitative approaches for software management, engineering and quality assurance by developing models and metrics for the software development process and product.

Dr. Basili has authored over 90 papers. In 1982, he received the Outstanding Paper Award from the IEEE Transactions on Software Engineering for his paper on the evaluation of methodologies.

He was Program Chairman for several conferences including the 6th International Conference on Software Engineering. He serves on the editorial boards of the *Journal of Systems and Software* and the IEEE *Transactions on Software Engineering* and is currently Editor-in-Chief of *TSE*. He is a member of the Board of Governors of the IEEE Computer Society.

THE 'SOFTWARE FIRST' SYSTEM DEVELOPMENT METHODOLOGY

Craig Brooks
U. S. Army
CECOM
Center for C3
Systems
Ft. Monmouth

Edward J. Gallagher, Jr.
U. S. Army CECOM
Center for Software
Engineering
Ft. Monmouth

David Preston
IIT Research
Institute
4550 Forbes Blvd
Lanham, MD 20706

ABSTRACT

Software first has been a theoretical goal in computer science since the early 1970s when software costs began to exceed hardware costs. However, with each machine having a unique assembly language, and with the tremendous variability in high-order language compilers from machine to machine, software remained machine dependent. The portability of Ada software finally makes software first a feasible concept. A research effort to define a software first system development methodology has recently been initiated. This paper describes the background of the software first philosophy, the approach being taken to define the methodology, and the objectives of the anticipated life-cycle phases.

INTRODUCTION

The classical method of system development is to choose the hardware for the system, fit the software to the system, then add hardware components and both add and alter software components to make the system work. By the time the system is fielded, the hardware is several years old and no longer state of the art. The software design has been altered to fit the hardware of choice and has therefore become both machine-dependent and difficult to maintain. And, the schedule has slipped due to incompatibilities between the hardware and software. The negative effects of this approach are most pronounced during the maintenance phase of the life cycle. In response to this situation, the U. S. Army's Communication Electronics Command (CECOM) Center for Software Engineering and IIT Research Institute have recently initiated an effort to define a software first system development methodology. The objectives of the proposed software first methodology are to produce more reliable, less costly systems by avoiding the shortcomings of the current approach. The intent of the methodology is to allow software to be developed prior to selection of the target computer; to permit development and initial testing on a programmer-friendly host computer; to determine functional and operational software changes by executing the software on the host by end users; to implement the necessary changes on the host system; to define the hardware requirements by instrumenting the host system; and to enhance the confidence in the software component of the system prior to porting the software to the target hardware. The benefits of this approach will hopefully be realized both during development and, more importantly, during the operations phase of the system life cycle. When proposed changes come in on a fielded system, their feasibility and impact can be assessed using the host environment prior to updating the targeted version of the system. Beyond the technical effects on system development, this approach will hopefully introduce more competition into the contracting process since intimate knowledge of a system-specific target hardware configuration will no longer be necessary to develop or maintain the software. The other anticipated major effect will be on training. Training will be started earlier, and lessons learned from the training process will be fed into the development process.

With the development and standardization of Ada, Department of Defense software has become more portable, less machine dependent, and therefore a software first development methodology now appears feasible. It is assumed here that the portability of Ada will minimize the effort required to retarget the software developed. It is also assumed that any changes that must be made to the software will be made to the software running in the host environment. Further, the development tools will be available at the site performing post-deployment software support. Finally, with decisions being made with respect to quality software

Support for this research was provided by RADC Contract Number F30602-86-C-0111.

engineering practices, requirements will be more easily maintained and traced and 'requirements creep' minimized.

This initial phase of the study will define the software first development methodology and identify the primary areas of concern to be explored during the next phase. A specific concern, for example, is the development of acceptance criteria which are measurable in the host environment and which imply specific levels of performance in the target environment. Other concerns which will be addressed during the course of the study include the degree to which Ada is portable, the compatibility of various development techniques with DoD-STD-2167, and potential interaction effects between the Ada language, real-time system development, and the software first development methodology. By first addressing the most challenging systems, real-time, it is believed that solutions will be more easily ported to other system types.

Inputs to this phase include the results of experimental applications of the software first system development methodology performed by CECOM's Center for Software Engineering. These applications have included systems developed in Fortran, C, and BASIC, and include a current project being developed in Ada. Also, the planned redevelopment of the Enhanced Position Location and Reporting System (EPLRS) will be performed using the software first system development methodology. Additional inputs include the results of ongoing studies on the use of Ada in real-time systems and on Ada runtime environments, existing system development methodologies, military standards that govern DoD software development efforts, and techniques that address a particular life-cycle phase such as design techniques. This phase of the study will be completed in November 1988.

The balance of this paper provides a background of the origins of the software first concept and outlines the approach of the study and the objectives of each anticipated life-cycle phase.

BACKGROUND

By the early 1970s it had been determined that software had become "the tall pole in the tent." The Air Force budget for fiscal year 1972 indicated that between $1 billion and $1.5 billion was spent on software, approximately three times the cost of hardware for the same period [BOEH73]. Not only was software costing more than the hardware component of a system, but it was also believed to be more responsible for schedule slippages, cost overruns, operational penalties and performance penalties, as well as for complex embedded problems that surfaced after system fielding. In the one and a half decades since 1972, the problem has continued to grow and the software pole has become even taller.

Both the concept of software first and a software first development machine were proposed in the early 1970s, although both the concept and machine being proposed here are significantly different from their predecessors. The intent to bring system cost, schedule and performance under control is still the same but the approach can now be more ambitious. Although many of the problems highlighted earlier either remain or have grown in magnitude, a significant new tool exists which we believe can facilitate software first becoming a reality. That tool is the Ada programming language.

The software first concept of the early 1970s was not literally a software first approach but rather a concurrent software and hardware development approach [FLEI74]. The approach called for an iterative system design process with several iterations possible between logical levels. Once the design had been completed, software development and hardware fabrication could be initiated. The theory proposed that final testing needed to be done on the actual system hardware and, therefore, the period for parallel development of hardware and software was between the completion of the system design process and final testing. The proposed software first concept is much more literal in its interpretation of the software development being first.

The software first machine proposed in the early 1970s was a generalized computer capable of simulating several computers and several computer configurations [BOEH73]. Software would be developed by assuming a specific architecture and then determining its adequacy as the software was developed. This would require that the software first machine be able to emulate the particular machine of interest. Although various aspects of the architecture could be altered during the course of development, such as memory size and clock speed, the instruction set in which the software was being developed would obviously be fixed. The set of hardware options would then be limited to machines with the same instruction set. The proposed software first development methodology would not require the construction of a specific software first machine with simulation capabilities. Rather, any computer with Ada cross compilers would be a candidate host machine, with the potential targets being the set of computers for which cross compilers from the host exist.

Although the software first machine of the 1970s is more consistent with what is being proposed here than is the software concept of the 1970s, there are significant differences with the machines as well. Rather than a special purpose machine being constructed for software development, the focus would be on a machine with strong software development tools and Ada compilers with several potential targets. The technology exists and has been successfully demonstrated to have multiple Ada cross compilers utilize the same front end [DEBA86]. The advantages of this technology include having a consistent user interface, having compiler enhancements available for all cross compilers at the same time, and making the development of new cross compilers more feasible. The significance of Ada's portability in general and this cross compiler technology in particular is that software first machines already exist and, given today's technology, make software first a quite plausible approach to system development.

APPROACH

The methodology to be produced by this effort will be a truly software-oriented development methodology. It will also be one which will be usable in the near term and as much as possible avóid reinventing the wheel.

The intent of this methodology is to reduce total life-cycle costs by enhancing the flexibility, portability, and overall quality of the software developed. The methodology will be based on making software-oriented decisions without making compromises in software development to satisfy hardware constraints.

The methodology will be generic in the sense that it will not be oriented to a particular type of software, although the inputs to the methodology will be oriented toward complex, real-time systems such as are found in DoD mission-critical systems.

LIFE-CYCLE PHASE OBJECTIVES

The structure of the methodology to be developed will not be constrained to follow the structure of any existing methodology. However, the functions defined by various phases of the standard development cycle must be satisfied by any comprehensive methodology. Although the structure may not follow the outline below, the methodology produced will perform the indicated functions.

System Requirements Analysis

The software first methodology will detail a method for partitioning system requirements into software and hardware requirements. The determination as to whether a system requirement will be met by hardware or software (or in some cases a combination of the two) will be based on several key factors. These factors can be identified by answering the following questions:

> Has a similar or related requirement been successfully satisfied in software?
>
> Are the required software interfaces within the current state of the art?
>
> For a distributed system, is there a similar software architecture to the one being considered?
>
> Are the system timing requirements that must be satisfied by software realistic?

These questions will be addressed during system requirements analysis to determine both the feasibility of successfully satisfying the system requirements and the appropriate hardware/software split of requirements. Software prototyping will be utilized during systems requirements analysis to establish the feasibility of satisfying requirements in software. These prototypes will be used to establish the feasibility of requirements and will not be considered as models of how to satisfy the requirement.

Software Requirements Analysis

Once the software requirements have been identified from the system requirements, software requirements analysis will be performed. The software first methodology will provide an approach to software requirements analysis that will assure that the requirements are both achievable and sufficient to meet the needs of the customer. The high level feasibility of the requirements will be established in system requirements analysis, and a more specific feasibility study will be conducted during software requirements. During software requirements analysis, groups of requirements will be collectively prototyped to establish the feasibility of concurrently satisfying related and interdependent requirements. Also, the more complex requirements will be addressed using exploratory programming. Although the prototypes are used only to establish the existence of a potential solution, the exploratory programming solutions may be used to assist in defining the design of the system and approximating the algorithms to be implemented. The prototypes and modules developed using exploratory programming will be used as vehicles for clarifying requirements, with the intent being to minimize the probability of changing requirements later in the life cycle.

Design, Code, and Integration

The single most important aspect of this phase is that all of the capabilities of the development environment - the memory, the development tools, flexibility of configuration - are available both for development and to address the problems of the hardware/software interface. This may require the future development of a database of target hardware capabilities and characteristics.

The software first methodology will be developed to eliminate the constraints imposed on designing, coding, and integrating the system by the hardware configuration. The development configuration will obviously be determined; however, the target computer will not have been chosen during these phases. This will allow all decisions to be made based on the capabilities and limitations of the state of the art of software engineering and software technology. The development of the software architecture, the system's algorithms, and the software interfaces will all be based solely on the software requirements and the capabilities of software.

Integration and testing will have multiple sub-phases. Initial testing and software integration will begin using the development computer. Hardware to software integration will have a major shift from the current perspective: If the hardware to software interface creates a problem during integration, the hardware configuration will be altered to solve the problem. It is anticipated that a major area of concern during the hardware to software integration will be the runtime environment.

Target Configuration

Once software has been developed, tested, and integrated on the development computer, the selection of the target configuration will be finalized. Time-critical functions will be implemented and run on potential target configurations to assess the exact timing capabilities of each potential hardware configuration. A major challenge to be addressed at this phase of the life cycle involves the Ada runtime environments available for potential target machines. Runtime environments will differ in timing-related capabilities. Another major effect of runtime environments will be on memory requirements.

The process of determining if a hardware configuration satisfies system memory requirements will, necessarily, be iterative. A potential runtime environment may run on multiple target machines, and each potential target may have several runtime environments that it supports. An initial estimate of candidate runtime environments, or candidate target machines, will need to be established and the evaluation of a target runtime environment pair performed. The runtime environment and the target machine must be evaluated in tandem, not independently. Also, the evaluation must consider both timing and memory constraints on each potential configuration. Some guidance for the initial set of candidate configurations may be attainable from the prototyping and exploratory programming performed during the system and software requirements analyses.

Operations

Although the operations phase is not typically defined in a development methodology - other than to say that all major upgrades to a system will follow the development methodology - the benefits of the software first methodology should be realized during the operations phase. The software will be developed based on software requirements and using software engineering practices with no compromises to support previously identified hardware. Therefore, the code should reflect the standard attributes such as readability and understandability that are associated with reduced maintenance. However, the major saving may be realized through the ease of system hardware changes. Since the software first development methodology will produce machine-independent code, subsequent hardware changes will be much easier to implement.

CONCLUSIONS

This effort to define a software first system development methodology has just begun. We, therefore, currently have more promise than product to report. However, if Ada proves to be portable, we believe that the major hurdle to a software first approach to system development has been cleared. Portability is obviously not a simple "yes" or "no" issue, but rather a question of how little effort it requires to port software from one computer to another. Examples of Ada code, even complex code [LAND87], being easily ported exist and have been reported. Guidelines to enhance the portability of software have been recommended [SILV87]. The portability of Ada software is being confirmed, and this confirmation has positive implications for the feasibility of software first.

In addition, the early analysis of experimental applications of the software first system development methodology performed at CECOM's Center for Software Engineering indicate that, with forethought, even non-Ada software can be ported easily. The current, larger efforts being done in Ada will provide more direction for this system development methodology.

The DoD software crisis is real. Previously recommended approaches to this problem have focused on standardizing on computer hardware [GOLU82] or on standardizing on a limited number of architectures and high-order languages [MORA78]. A primary advantage of both of these approaches is that they limit the number of systems and development environments with which developers and maintainers must become familiar. The software first approach would realize this same benefit.

More recently recommended approaches to the software problem have focused on programmer productivity [BOEH87]. Since the Ada programming language is instrumental in software first, the indications of increased productivity with Ada [CAST87] are a positive indicator for increased productivity using software first.

Technical and managerial impediments to implementing the software first system development methodology exist, and some of these are yet to be identified. We believe that the positive side of the software first ledger far outweighs the negative side.

REFERENCES

[BOEH73] Boehm, Barry. "Software and Its Impact: A Quantitative Assessment," Datamation, Vol. 19, No. 5, May 1973.

[BOEH87] Boehm, Barry. "Improving Software Productivity," Computer, Vol. 20, No. 9, September 1987.

[CAST87] Castor, Virginia and David

Preston. "Programmers Produce More With Ada," Defense Electronics, June 1987.

[DEBA86] De Bartolo, Gil and Ron Richards. "The Back-End of a Multi-Target Compiler," Proceedings of the 4th National Conference on Ada Technology, March 19-20, 1986.

[FLEI74] Fleisher, Robert J. "Software-First System Design," *Compcon76*, February 24-26, 1976.

[GOLU82] Golubjatnikov, Ole. "Architecture, Hardware and Software Issues in Fielding the Next Generation DoD Processors," Proceedings of the 2nd AFSC Standardization Conference, December 1982.

[MORA78] Moralee, Dennis. "MIL-SPEC Computers - Building the Hardware to Fit the Software," Electronics and Power, August 1978.

[SIVL87] Sivley, Karen E. "Experience and Lessons Learned in Transporting Ada Software," Proceedings of the Joint Ada Conference Fifth National Conference on Ada Technology and Washington Ada Symposium, March 16-19, 1987

BIOGRAPHICAL SKETCHES

Mr. Craig Brooks is a member of the engineering staff of the Ft. Monmouth Center for Command, Control, and Communication Systems (formerly Communications/ADP Laboratory), where he has worked since 1964. He is presently active in software cost analysis, the PM common hardware/software project, the PM, ADDS/TIDS projects, and the Software First System Development Program. In his career, Mr. Brooks has been an ADP consultant on various Army, Navy, and Marine Corps embedded computer projects. He was the government project engineer on the Tactical Fire Direction System (TACFIRE), the Military Computer Family (MCF) program, and a number of computer memory technology programs. Mr. Brooks was active in the early development of the Ada language and served as the government in-plant representative on the Ada Language System development program. Mr. Brooks received the B. E. E. degree from Clarkson College of Technology in 1963.

Mr. Edward J. Gallagher, Jr. is currently a Branch Chief in CECOM's Center for Software Engineering and he is responsible for their research and development efforts covering software reuse and Ada applied to real-time systems and the associated area of runtime environments. In his previous assignment with Project Manager PLRS/TIDS he was responsible for all the software for the PLRS/JTIDS hybrid, a complex position location/communication system. He has also served as the chairman of the STARS Human Resources Area Coordinating Team and has written and reviewed Ada and computer resources policy for the command. He received his B. S. in Electrical Engineering from Carnegie Mellon University, and an M. S. in Management Science from Fairleigh Dickinson.

Dr. David Preston is a Senior Software Engineer with IIT Research Institute and an adjunct faculty member of the Computer Science Department of the University of Maryland. His specific research interests are the use of Ada for secure systems, real-time application issues, and runtime environment evaluation techniques and criteria. He is a member of IEEE, the IEEE Computer Society, and the IEEE Technical Committee on Software Engineering. He holds a B. S. in Earth and Space Science from Clarion State College, an M. S. in Mathematics from Ohio University, and a Ph. D. in Mathematics Education and an M. S. in Computer Science from the University of Maryland.

Graphical Database Management and Its Impact on Ada System Design

Keith E. Bernard & Daniel M. Butler
Telos Federal Systems
1315 Directors Row
Fort Wayne, Indiana 46808

ABSTRACT

Various examples of Graphical Database Management Systems (DBMS) developed for use on the Apple Macintosh computer are currently available to the public. Because these products are database applications, they present the possibility of collecting the totality of information associated with a software system within one application. Also, because these products were developed for the Macintosh, they allow a user to use its graphic capabilities. Simply stated, these products readily lend themselves to the task of associating pictures to words and thus pictures to Ada code.

1. Benefits of Using a Graphical DBMS:

A system that is developed using a Graphical DBMS has the capacity to graphically represent the static structural levels described in DoD-STD-2167 as: Computer Software Configuration Item (CSCI), Top Level Computer Software Component (TLCSC), Low Level Computer Software Component (LLCSC), and Unit (Figure 1). In addition, such a system allows the graphical representation of data/control flow diagrams for each of these static structural levels. Furthermore, it has the capability of associating requirements and interfaces to each of these levels, as well as presenting actual Ada code at the Unit level (Figure 2).

Importance of graphical representation of data/control flow.

In the development of Ada systems, as is true of any system, a clear understanding of data/control flow is required to ensure that the design phase results in a complete system design. Graphical representation of data/control flow helps to reduce the possibility of producing a system design that is not complete.

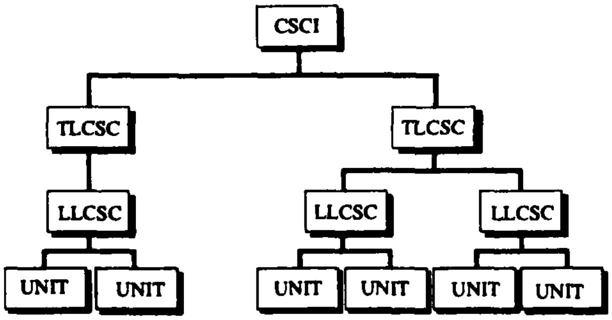


Figure 1. Static Structural Levels as Described in DoD-STD-2167

In some Ada software development environments, for example, it may be costly to discover that a procedure in a package that many other packages depend upon requires an additional parameter to be added to its parameter list. Changing package specifications during the coding phase, the phase where most design problems are currently identified, is both frustrating and wasteful. Graphical representation of data/control in the system design effort would better present an overall picture of the system. This overall picture would help to ensure a complete design as well as to promote understandability of the system's requirements.

Importance of associating requirements with static structural levels.

Requirements as described here may be any requirements associated with a system. If MIL-STD-490A is used, the requirements may be in the form of B5 paragraphs. The result of associating requirements to each of the static structural levels will be an improved traceability of requirements throughout the system.

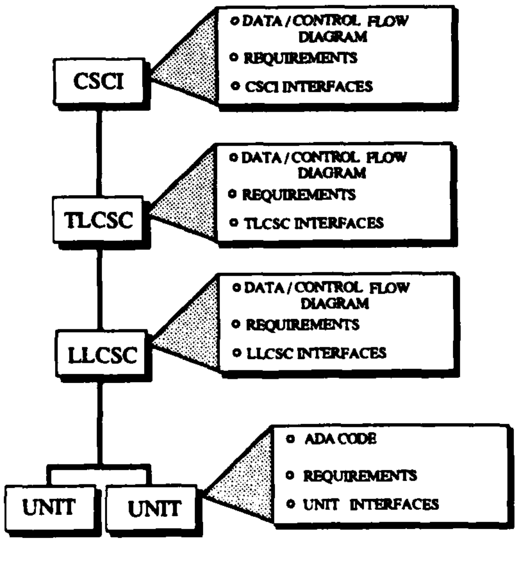


Figure 2. Information to be Associated with Each Static Structural Level

Importance of Ada code at the Unit level.

The presentation of Ada code at the Unit level of the static structure will provide a means of directly mapping data/control flow diagrams as well as Unit requirements and interfaces to that code. If the user of such a tool desires to view the Ada code associated with a particular procedure shown on the data/control diagram, he simply selects the area on the diagram corresponding to that procedure, and the Graphical DBMS will navigate to the file that contains the associated Ada code. Inversely, if the user is viewing Ada code, the Graphical DBMS can easily navigate to the data/control diagram, requirements, or interfaces which correspond to that code. The result is a more intimate knowledge of how the code for a particular Unit fits into a broader view of the entire system.

2. Developing Data/Control Flow Diagrams Using a Methodology Suited for an Ada System

There does not seem to be a single design methodology suited to the development of an Ada system which will fully facilitate the software engineering principles of abstraction, information hiding, modularity, localization, uniformity, confirmability and completeness. Although Object Oriented Design is a powerful, proven design tool which promotes these principles as well as the reusability of software, it fails to ensure a complete system design. Additionally, while design approaches may provide completeness, they tend to lack support of the engineering principles stated above and fail to promote software reuse.

In order to obtain the goal of producing a system that is modifiable, efficient, reliable, and understandable, by fully facilitating the principles of software engineering, one must merge the benefits of more than a single design methodology. More specifically, one must tailor the Object Oriented Design methodology to enforce the completeness of system design by combining it with those aspects of the structured analysis approach that result in a full understanding of data/control flow and of system/subsystem requirements. This full range of information can be captured, managed, and manipulated by the incorporation of a Graphical DBMS.

Diagrams produced through recursive Object Oriented Design.

Some implementations of an OOD methodology merely recommend the use of Data/Control Flow diagrams to aid in the analysis and clarification of the requirements. Using the above stated method of design, OOD, and Structured Analysis, the emphasis is placed on the analysis of the requirements during the design phase. When using such a design method, Data/Control Flow diagrams are no longer recommended – they become required.

Data/Control Flow diagrams are produced during the analysis and clarification of the requirements phase of the OOD process. This phase is used to gather and analyze information necessary for solving a particular problem of the system. The Data/Control Flow diagrams are not intended to enforce a particular design, but rather to ensure a complete understanding of the problem. The diagrams will organize the requirements and the OOD process will drive the design of the system. The goal is to have the Data/Control Flow diagrams ensure a clearer understanding of the problem to be solved, rather than communicating the design intent.

Lower levels of Data/Control Flow diagrams are produced with each recursion of the OOD process. Each recursion prompts a lower level of abstraction, and thus, a lower structural level.

The System Development Process incorporating a Graphical DBMS is summarized below:

1. Develop the CSCI.
   - State the problem of the CSCI.
   - Analyze and Clarify the problem using Data/Control Flow Diagrams identifying data within, to, and from the CSCI.
   - Write the OOD paragraph, identifying objects and operations.
   - Group the objects and operations.
   - Make design decisions.
   - Identify interfaces.
   - Identify operations that require recursion. These become TLCSCs.
2. Develop the TLCSC.
   - State the problem of the TLCSC.
   - Analyze and clarify the problem using Data/Control Flow Diagrams identifying data within, to, and from the TLCSC.
   - Write the OOD paragraph, identifying objects and operations.
   - Group objects and operations.
   - Make design decisions.
   - Identify interfaces.
   - Identify operations that require recursion. These become LLCSCs.
3. Develop the LLCSC.
   - State the problem of the LLCSC.
   - Analyze and clarify the problem using Data/Control Flow Diagrams identifying data within, to, and from the LLCSC.
   - Write the OOD paragraph, identifying objects and operations.
   - Group objects and operations.
   - Make design decisions.
   - Identify interfaces.
   - Identify operations that require recursion. These become Units.

4. Develop the Unit.

    - Make design decisions.
    - Identify interfaces.
    - Implement solution.

Recursive OOD, if used in this manner, will produce a more complete design and packages with levels of abstraction which correspond to the high, mid, and low level of the static structure.

3. Ada System Development Using a Graphical DBMS

The following is the presentation of an example which illustrates the development process stated previously:

1. Develop the CSCI.

    - State the problem of the CSCI.

      For this step, write a single sentence describing the problem to be solved:

      The problem is to implement a fire support activity.

    - Analyze and clarify the problem using a Data/Control flow diagram to identify data within, to, and from the CSCI (Figure 3).

      The purpose of this step is to analyze and clarify the problem using Data / Control flow diagrams in order to produce a consistent, complete OOD paragraph.

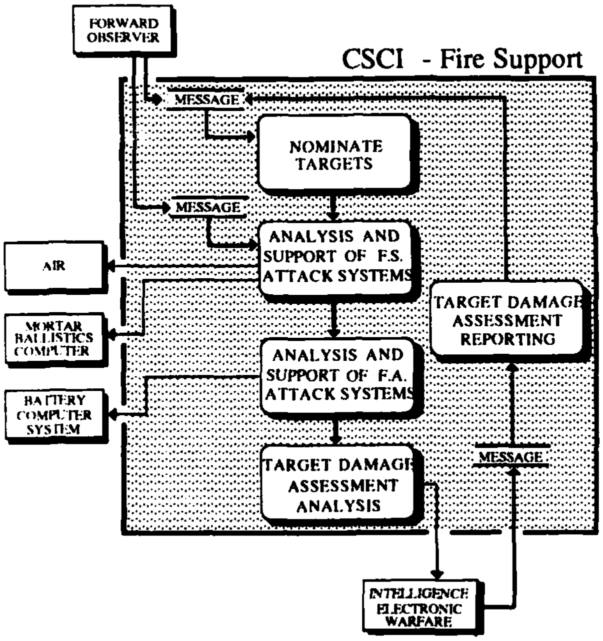

FORWARD OBSERVER
CSCI - Fire Support
MESSAGE
NOMINATE TARGETS
MESSAGE
ANALYSIS AND SUPPORT OF F.S. ATTACK SYSTEMS
AIR
MORTAR BALLISTICS COMPUTER
BATTERY COMPUTER SYSTEM
TARGET DAMAGE ASSESSMENT REPORTING
ANALYSIS AND SUPPORT OF F.A. ATTACK SYSTEMS
MESSAGE
TARGET DAMAGE ASSESSMENT ANALYSIS
INTELLIGENCE ELECTRONIC WARFARE

Figure 3. Data/Control Flow Diagram

It is very important that, during this step, a consistent level of abstraction is obtained. The Data/Control flow diagram for this level should contain only enough information necessary to solve the problem at the highest level. Finally, the Data/Control flow diagram should identify the external interfaces of the CSCI.

- Write the OOD paragraph, identifying objects and operations.

  The paragraph, to identify objects and operations, should be written to reflect the abstraction level identified in the Data/Control flow diagram:

  Nominate eligible targets for attack. Select and task fire support (FS) systems to attack the targets. If an FS system is not selected, select and task units with field artillery (FA) attack assets to attack the target. Also determine if target damage assessment (TDA) is required and if damage requirements were met. If damage requirements were not met, send the target to be renominated. Terminate fire support activities.

  The operations are:

  NOMINATE ELIGIBLE TARGETS FOR ATTACK
  SELECT AND TASK FS ATTACK SYSTEMS
  SELECT AND TASK FA ATTACK ASSETS
  DETERMINE IF TDA IS REQUIRED
  DETERMINE IF DAMAGE REQUIREMENTS WERE MET
  TERMINATE FIRE SUPPORT ACTIVITIES

- Group the objects and operations.

  The operations
  NOMINATE ELIGIBLE TARGETS FOR ATTACK
  SELECT AND TASK FS ATTACK SYSTEMS
  SELECT AND TASK FA ATTACK ASSETS
  DETERMINE IF TDA IS REQUIRED
  DETERMINE IF DAMAGE REQUIREMENTS WERE MET
  TERMINATE FIRE SUPPORT ACTIVITIES
  do not act on any identified objects.

- Make design decisions.

  The CSCI will consist of a main routine, FIRE SUPPORT, and six subroutines which correspond to the operations identified by the CSCI OOD paragraph.

- Identify interfaces.

  Figure 4 shows the interfaces within the CSCI.

- Identify operations that require recursion. These become the TLCSCs.

  As a result of the high level of abstraction of this CSCI paragraph, the implementation of any of the identified operations is prevented. This does not imply that the OOD process was ineffective for this CSCI. OOD in this case did not produce the desired packages with groupings of objects and operations, but rather an operational (functional) decomposition. This decomposition identifies TLCSCs. Therefore, OOD must be reapplied to each operation identified in the CSCI paragraph to establish more detail. Thus, each operation will become a TLCSC.

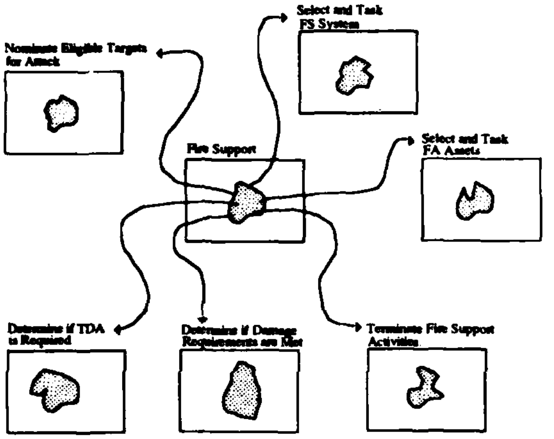
2. Develop the TLCSC.

- State the problem of the TLCSC. For the purpose of illustration, the "NOMINATE TARGETS FOR ATTACK" TLCSC will be developed.

  Write a single sentence describing the problem to be solved:

  The problem is to Nominate Eligible Targets for Attack.

- Analyze and clarify the problem using a Data/Control flow diagram to identify data within, to, and from the TLCSC (Figure 5).



Select and Task FS System
Nominate Eligible Targets for Attack
Fire Support
Select and Task FA Assets
Determine if TDA is Required
Determine if Damage Requirements are Met
Terminate Fire Support Activities

Figure 4. Interfaces of the FIRE SUPPORT CSCI

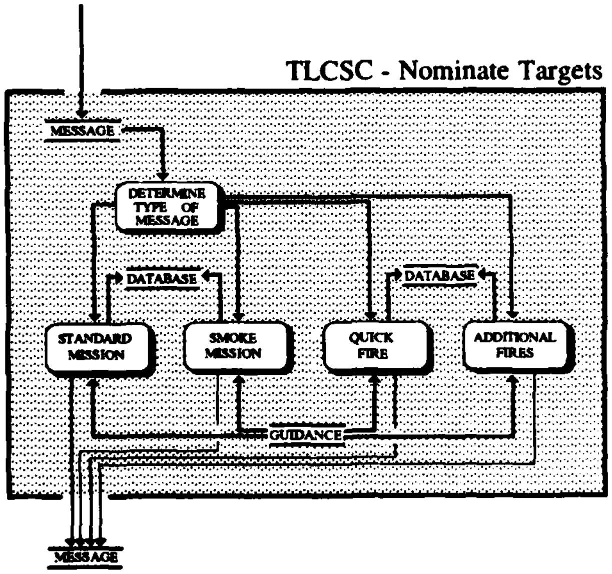


Figure 5. TLCSC Data/Control Flow Diagram

- Write the OOD paragraph, identifying objects and operations:

  Receive a message. Identify the type of message to be processed. If the message is a standard mission, process the standard mission. If the message is a smoke mission, process the smoke mission. If the message is a quick fire, process the quick fire. If the message is a request for additional fires, process the request for additional fires.

  The objects are:

  MESSAGE

  The operations are:

  RECEIVE A MESSAGE
  DETERMINE THE TYPE OF MESSAGE
  PROCESS STANDARD MISSION
  PROCESS SMOKE MISSION
  PROCESS QUICK FIRE MISSION
  PROCESS REQUEST FOR ADDITIONAL FIRES

- Group the objects and operations.

  The operations
  RECEIVE A MESSAGE
  DETERMINE THE TYPE OF MESSAGE
  act on the object
  MESSAGE.

  The operations
  PROCESS STANDARD MISSION
  PROCESS SMOKE MISSION
  PROCESS QUICK FIRE MISSION
  PROCESS REQUEST FOR ADDITIONAL FIRES
  do not act on any identified object.

- Make design decisions.

  The NOMINATE ELIGIBLE TARGETS subroutine identified when developing the CSCI will become Nominate_Eligible_Targets_Message_Processor_Package and will contain the task Target_Message_Processor_Task and the operations Process_Standard_Mission, Process_Smoke_Mission, Process_Quick_Fire_Mission, and Process_Request_For_Additional_Fires. The package Message_Package will also define the type Message_Type as well as the operations Receive_A_Message and Determine_Type_Of_Message.

- Identify interfaces.

  Figure 6 shows the interfaces within the TLCSC.

- Identify operations that require recursion. These become the LLCSCs.

  The operations
  PROCESS STANDARD MISSION
  PROCESS SMOKE MISSION
  PROCESS QUICK FIRE MISSION

PROCESS REQUEST FOR ADDITIONAL FIRES

all require recursion since they require further detail to implement. These operations will become the LLCSCs.

3. Develop the LLCSC.

- State the problem of the LLCSC. For this example, the "PROCESS STANDARD MISSION" LLCSC will be developed.

  Write a single sentence describing the problem to be solved.

  The problem is to process a Standard Mission.

- Analyze and clarify the problem using a Data/Control flow diagram to identify data within, to, and from the LLCSC (Figure 7).

- Write the OOD paragraph, identifying objects and operations.

  Using guidance, determine if the message should be processed or sent out. If the message is to be processed, determine if the message meets target classification guidance. If the message is a target, assign a priority to the target. Using guidance, determine if the message should be routed to another agency or if the target is a duplicate of another target. If the message is not routed and the target is not a duplicate of any other target, nominate the target for attack.

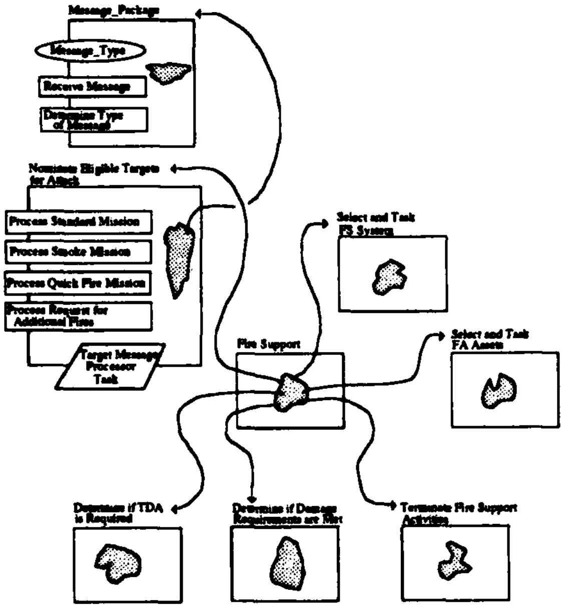

Message_Package
Message_Type
Receive Message
Determine Type of Message
Nominate Eligible Targets for Attack
Process Standard Mission
Process Smoke Mission
Process Quick Fire Mission
Process Request for Additional Fires
Target Message Processor Task
Fire Support
Select and Task FS System
Select and Task FA Assets
Determine if TDA is Required
Determine if Damage Requirements are Met
Terminate Fire Support Activities

Figure 6. Interfaces of the NOMINATE TARGETS TLCSC

The objects are:
MESSAGE
TARGET
GUIDANCE

The operations are:
DETERMINE IF MESSAGE SHOULD BE PROCESSED
DETERMINE IF MESSAGE MEETS CLASSIFICATION GUIDANCE
IF THE MESSAGE IS A TARGET
ASSIGN A PRIORITY
DETERMINE IF THE MESSAGE SHOULD BE ROUTED TO AGENCY
DETERMINE IF THE TARGET IS A DUPLICATE
NOMINATE THE TARGET FOR ATTACK

- Group the objects and operations.

  The operations
  DETERMINE IF MESSAGE SHOULD BE PROCESSED
  DETERMINE IF MESSAGE MEETS CLASSIFICATION GUIDANCE
  IF THE MESSAGE IS A TARGET
  ASSIGN A PRIORITY
  DETERMINE IF THE MESSAGE SHOULD BE ROUTED TO AGENCY
  DETERMINE IF THE TARGET IS A DUPLICATE
  NOMINATE THE TARGET FOR ATTACK
  act on the object
  MESSAGE.

  The object GUIDANCE will be grouped with no operations.

- Make design decisions.

  The operations Determine_If_Message_Should_Be_Processed, Assign_Priority, Determine_If_Message_Meets_Classification_Guidance, Nominate_Target,

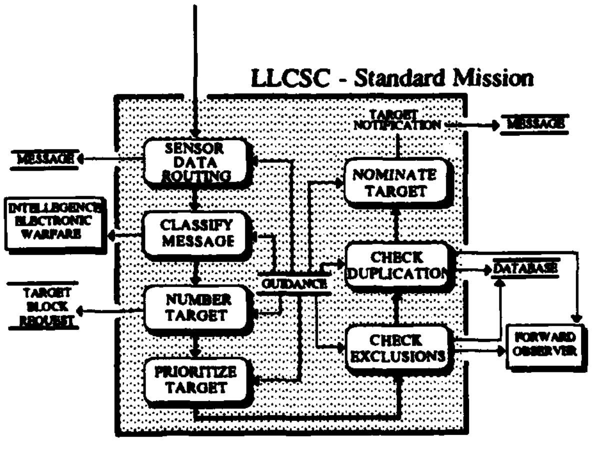


Figure 7. LLCSC Data/Control Flow Diagram

Determine_If_Message_Should_Be_Routed_To_Agency, and Determine_If_Message_ Is_Duplicate will be added to the Message_Package. In addition, the Guidance_Packace will define the type Guidance_Type and will contain the operations on that type when they are identified.

- Identify interfaces.

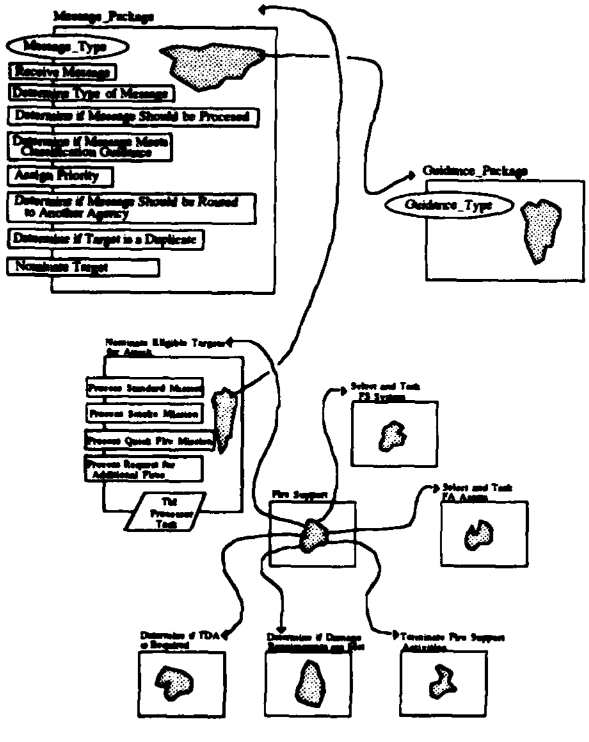
  Figure 8 shows the visible interfaces within the LLCSC.

4. Develop the Unit.

   - For this example, the "CLASSIFY MESSAGE" Unit will be developed.

   - Make design decisions.

     The Guidance_Type will consist of a Target_Decay_Time_Table and the operation to Get_Target_Decay_Time will be added to the Guidance_Package. The Guidance_Type will also consist of a Sensor_Accuracy_Reliability_Table and the operations to Get_Sensor_Accuracy and Get_Sensor_Reli-ability. The operation Field_Is_Present will be added to the Message_Package to determine if a specified field in the message is present. These types and operations have been identified from the Classify_Message requirements to implementing the Unit.



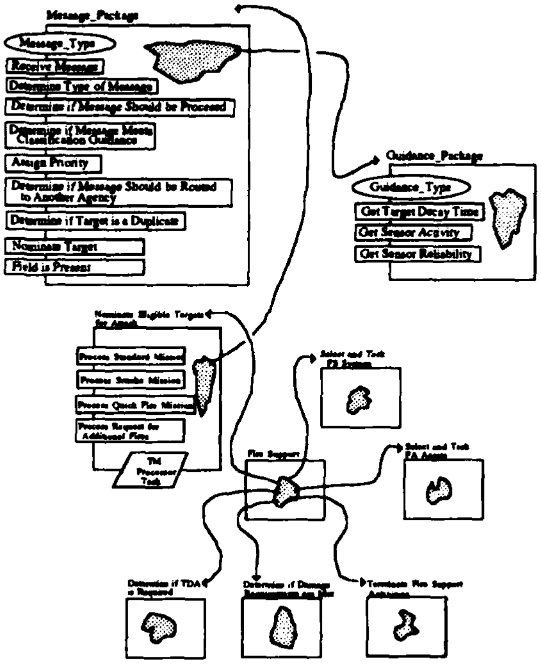
Message_Package
Message_Type
Receive Message
Determine Type of Message
Determine if Message Should be Proceesd
Determine if Message Meets Classification Guidance
Assign Priority
Determine if Message Should be Routed to Another Agency
Determine if Target is a Duplicate
Nominate Target
Guidance_Package
Guidance_Type
Nominate Eligible Targets for Attack
Process Standard Mission
Process Smoke Mission
Process Quick Fire Mission
Process Request for Additional Fires
TM Processor Task
Fire Support
Select and Task FS System
Select and Task FA Assets
Determine if TDA is Required
Determine if Damage
Terminate Fire Support Activities

Figure 8. Interfaces of the STANDARD MISSION LLCSC

- Identify interfaces.

  Figure 9 shows the visible interfaces within the Unit.

- Implement solution.



Message_Package
Message_Type
Receive Message
Determine Type of Message
Determine if Message Should be Proceesd
Determine if Message Meets Classification Guidance
Assign Priority
Determine if Message Should be Routed to Another Agency
Determine if Target is a Duplicate
Nominate Target
Field is Present
Guidance_Package
Guidance_Type
Get Target Decay Time
Get Sensor Accivity
Get Sensor Reliability
Nominate Eligible Targets for Attack
Process Standard Mission
Process Smoke Mission
Process Quick Fire Mission
Process Request for Additional Fires
TM Processor Task
Fire Support
Select and Task FS System
Select and Task FA Assets
Determine if TDA is Required
Determine if Damage
Terminate Fire Support Activities

Figure 9. Interfaces of the CLASSIFY MESSAGE UNIT

4. Implementation of a System Developed Using a Graphical DBMS

The navigation through a system developed using a Graphical DBMS is depicted in Figure 10. (Note the menu selections available at each level.) From the CSCI Data/Control flow diagram, a TLCSC is selected and the appropriate menu appears. The TLCSC Data/Control Flow Diagram option is then chosen from the menu. This process is repeated until the desired structural level is reached, in this case, the Unit level. Both the Ada code and requirements, as well as the interfaces corresponding to any unit, can then be viewed or edited at will. In addition to this feature, all levels of the structure offer supplemental information concerning any data flow or distinct entity within that level.

The following example presents one scenario which puts to use the system developed using Graphical DBMS in the sections above. In this particular scenario, the user wishes to view the Ada code, requirements, and interfaces of the Unit CLASSIFY MESSAGE.

> The user selects the NOMINATE TARGETS TLCSC area of the CSCI Data/Control flow diagram (Figure 11). As a result of this selection, the TLCSC Main Menu appears and prompts the user for a selection. The TLCSC Data/Control Flow Diagram option is then chosen from the menu (Figure 12).

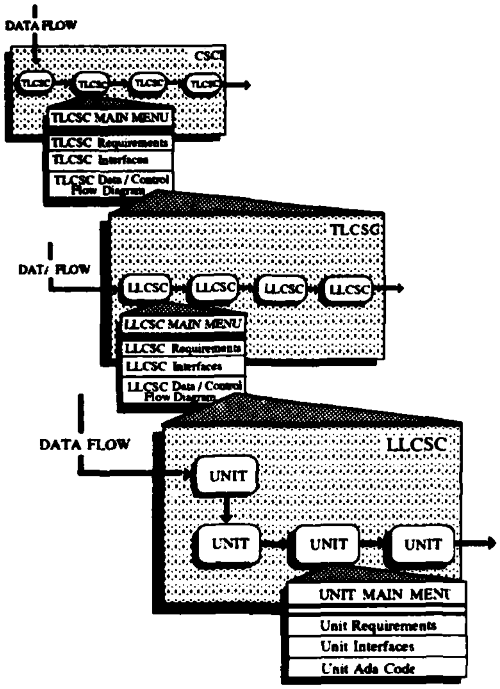


Figure 10. Navigation Through the Static Structural Levels

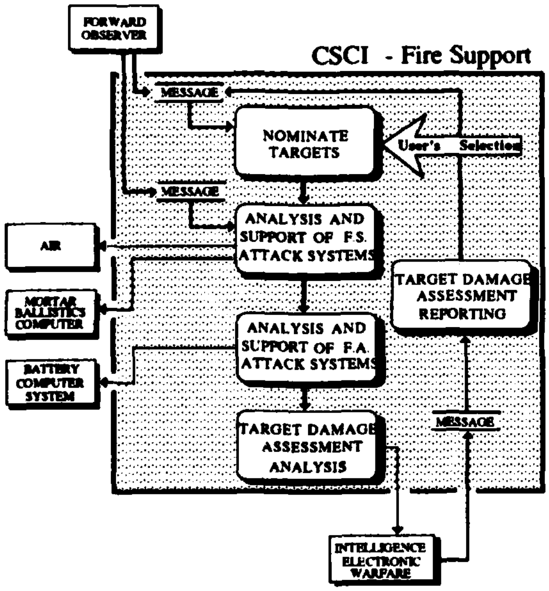

FORWARD OBSERVER
CSCI - Fire Support
MESSAGE
NOMINATE TARGETS
User's Selection
MESSAGE
ANALYSIS AND SUPPORT OF F.S. ATTACK SYSTEMS
AIR
MORTAR BALLISTICS COMPUTER
TARGET DAMAGE ASSESSMENT REPORTING
ANALYSIS AND SUPPORT OF F.A. ATTACK SYSTEMS
BATTERY COMPUTER SYSTEM
TARGET DAMAGE ASSESSMENT ANALYSIS
MESSAGE
INTELLIGENCE ELECTRONIC WARFARE

Figure 11. User's Selection from the CSCI Data/Control Flow Diagram

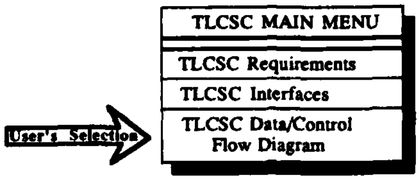


Figure 12. User's Selection from the TLCSC Main Menu

The user then selects the STANDARD MISSION LLCSC area of the TLCSC Data/Control flow diagram (Figure 13). Again prompted for a decision via the LLCSC Main Menu, the LLCSC Data/Control Flow Diagram option is chosen (Figure 14). This choice leaves the user at the Data/Control flow diagram for the STANDARD MISSION LLCSC.

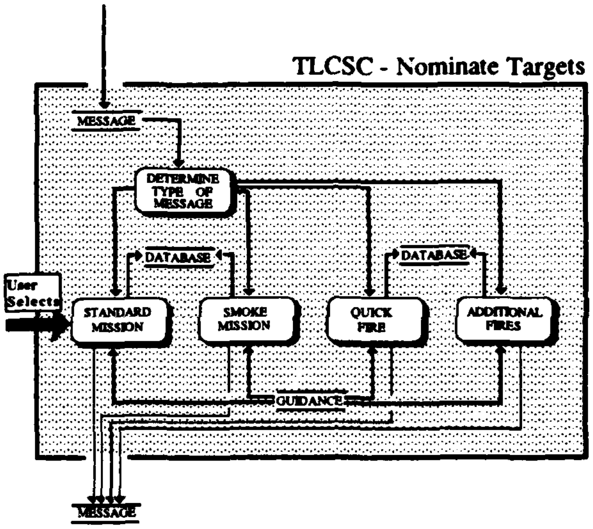


Figure 13. User's Selection from the TLCSC Data/Control Flow Diagram

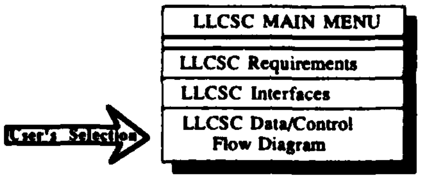


Figure 14. User's Selection from the LLCSC Main Menu

As stated earlier, all levels of the structure offer supplemental information concerning any data flow or distinct entity within that level. In order to stress this point, Figures 15 through 20 highlight the effects of selecting such items at the LLCSC level (For clarity, only a portion of the STANDARD MISSION LLCSC Data/Control flow diagram is shown in these figures). Figure 15 reflects the user's selection of the data flow from the GUIDANCE entity to the CLASSIFY MESSAGE Unit area. The results of this selection can be seen in Figure 18. Similarly, Figure 16 reflects the user's selection of the data flow between the Units SENSOR DATA ROUTING and CLASSIFY MESSAGE. These results can be seen in Figure 19. Finally, Figure 17 reflects the user's selection of the data flow from the CLASSIFY MESSAGE Unit to the entity INTELLIGENCE ELECTRONIC WARFARE. The results of this selection can be seen in Figure 20.

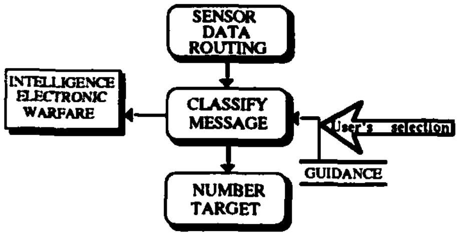


Figure 15. User's Selection of the Data Flow from GUIDANCE to CLASSIFY MESSAGE

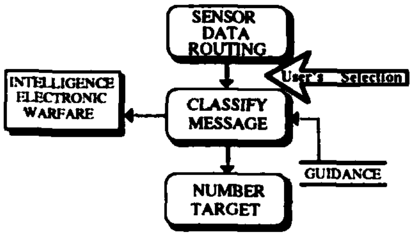


Figure 16. User's Selection of the Data Flow from SENSOR DATA ROUTING to CLASSIFY MESSAGE

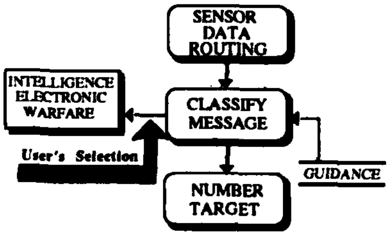


Figure 17. User's Selection of the Data Flow from CLASSIFY MESSAGE to INTELLIGENCE ELECTRONIC WARFARE

Guidance Information

Target Characteristics Guidance

- Used to assign a decay time to a target given a target type and target subtype.
- Contains:
  Target_Type
  Target_Subtype
  Target_Decay_Time

Sensor Accuracy Guidance

- Used to determine the accuracy of a specific sensor system.
- Contains:
  Sensor_Type
  Sensor_Accuracy

Sensor Reliability Guidance

- Used to determine the reliability of a specific sensor system.
- Contains:
  Sensor_Type
  Sensor_Reliability

Target Selection Criteria Guidance

- Used to define the minimum allowable values to classify a message as a target.
- Contains:
  Minimum_Decay_Time
  Minimum_Sensor_Reliability
  Minimum_Sensor_Accuracy
  Addressee_For_Non_Targets

Figure 18. Results of the User's Selection of the Data Flow from GUIDANCE to CLASSIFY MESSAGE

Message Information

- Message sent from SENSOR DATA ROUTING operation.
- Contains:
  Target_Decay_Time
  Target_Type
  Target_Subtype
  Sensor_Accuracy
  Sensor_Reliability
  Time_Target_Sensed

Figure 19. Results of the User's Selection of the Data Flow from SENSOR DATA ROUTING to CLASSIFY MESSAGE

Message to Intelligence Electronic Warfare

- Sent from CLASSIFY MESSAGE
- Contains:
  Sensor_Data_Information

Figure 20. Results of the User's Selection of the Data Flow from CLASSIFY MESSAGE to INTELLIGENCE ELECTRONIC WARFARE

When the user finally selects the CLASSIFY MESSAGE area of the LLCSC Data/Control Flow diagram, the Unit Main Menu appears. This menu allows mobility to both the Unit requirements and Ada code, as well as the Unit interfaces. A summary of all Unit Main Menu selections and results is presented in Figures 21 through 25. (Refer to Figure 9 to view the results of the user's selection in Figure 22.)

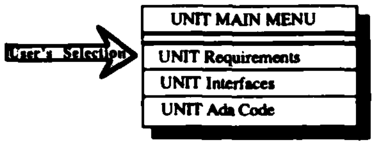


Figure 21. User's Selection of UNIT Requirements from the UNIT MAIN MENU

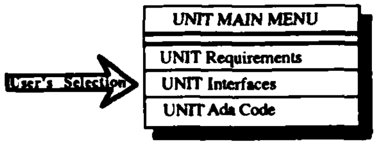


Figure 22. User's Selection of UNIT Interfaces from the UNIT MAIN MENU

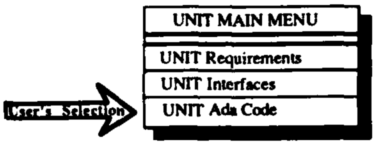


Figure 23. User's Selection of UNIT Ada Code from the UNIT MAIN MENU

CLASSIFY MESSAGE Unit Requirements

- The procedure Classify_Message shall update the Decay_Time field in the message with the decay time provided by the Target Characteristics Guidance whose target type and target subtype match the Target_Type and Target_Subtype specified in the message.
- If Sensor_Accuracy is not present in the message, Classify_Mesage shall update the Sensor_Accuracy field in the message with the sensor accuracy provided by the Sensor Accuracy Guidance whose sensor type matches the Sensor_Type specified in the message.
- If Sensor_Reliability is not present in the message, Classify_Message shall update the Sensor_Reliability field in the message with the sensor reliability provided by the Sensor Reliability Guidance whose sensor type matches the Sensor_Type specified in the message.
- If (Current_Time - Time_Target_Sensed) is greater than the target decay time specified in the Target Selection Criteria Guidance, the message shall be classified a non-target.
- If the Sensor_Accuracy field in the message is greater than the sensor accuracy specified in the Target Selection Criteria Guidance, the message shall be classified a non-target.
- If the Sensor_Reliability field in the message is greater than the sensor reliability specified in the Target Selection Criteria Guidance, the message shall be classified a non-target.
- If the message has been classified a non-target, Classify_Message shall send the message to the addressee specified in Target Selection Criteria Guidance. Otherwise, it will transfer control to Number_Target.

Figure 24. Results of the User's Selection of UNIT Requirements from the UNIT MAIN MENU

```
with Guidance_Package;
with Calendar;
separate Message_Package

procedure Classify_Message
 (
 Message                      : in out Message_Type;
 Guidance                     : in     Guidance_Package.Guidance_Type;
 Message_Is_A_Target            :    out Boolean
 )
 is

begin

 Current_Time  : Calendar.Time;
 Age_Criteria   : Calendar.Time;

--| Assign target decay time
  Message.Decay_Time
    := Guidance_Package.
       Get_Target_Decay_Time
          (
          Target_Type     => Message.Target_Type,
          Target_Subtype  => Message.Target_Subtype
          );

 if Field_Is_Present ( Sensor_Accuracy ) then
   null;  --| sensor accuracy is specified

 else
--| sensor accuracy is not specified
   Message.Sensor_Accuracy
    := Guidance_Package.
       Get_Sensor_Accuracy
         (
         Sensor_System => Sensor_System
         );

 end if;

  --| check sensor reliability
 if Field_Is_Present ( Sensor_Reliability ) then
   null;  --| sensor reliability is specified

 else

  --| sensor reliability is not specified
   Message.Sensor_Reliability
    := Guidance_Package.
        Get_Sensor_Reliability
          (
          Sensor System => Sensor System
          );

 end if;

 Current_Time := Calendar_Package.Get_Time;

 Age_Criteria
  := Guidance.Target_Selection_Criteria.Minimum_Decay_Time;

 if Message.Sensor_Accuracy
    > Target_Selection_Criteria.Source_Accuracy then
  Message_Is_A_Target := False;

 elsif Guidance_Package.Reliability_Type'Pos
       ( Message.Sensor_Reliability )
        > Guidance_Package.Reliability_Type'Pos
           ( Target_Selection_Criteria.Minimum_Reliability ) then
  Message_Is_A_Target := False;

 elsif Age_Of_Target > Age_Criteria then
  Message_Is_A_Target := False;

 else
  Message_Is_A_Target := True;

 end if;

end Classify_Message;
```

Figure 25. Results of the User's Selection of UNIT Ada Code from the UNIT MAIN MENU

KEITH E.BERNARD
Telos Federal Systems
1315 Directors Row
Fort Wayne, Indiana 46808

Mr. Bernard received his B.S. in Computer Science from Baylor University in 1984. He has been working as a Software Engineer on the Advanced Field Artillery Tactical Data System (AFATDS) for the past three years, and is a member of the Association for Computing Machinery (ACM) Ada Special Interest Group (SIG-Ada).

DANIEL M. BUTLER
Telos Federal Systems
1315 Directors Row
Fort Wayne, Indiana 46808

Mr. Butler received his B.S. in Management Science and Computer Systems (MIS) from Oklahoma State University in 1987. Since then, he has been involved in the development of the Elevated Target Aquisition System (ETAS), and the Advanced Field Artillery Tactical Data System (AFATDS).

METHODOLOGY FOR IMPLEMENTING ARTIFICIAL

INTELLIGENCE SYSTEMS IN ADA (R)

Bernard Abrams and Teresa Doran
Software Systems Department
Grumman Aircraft Systems Division
Bethpage, NY 11714

1 ABSTRACT

The languages most commonly used to prototype artificial intelligence (AI) systems are very different from the languages used to program avionics software. This paper proposes a methodology to develop artificial intelligence software for an avionics application despite this mismatch. The methodology involves prototyping in an AI language, then reimplementing in Ada using a library of AI functions. The pilot project used to verify this methodology is a maintenance diagnostic system prototyped in Lisp and reimplemented in Ada.

The proposed methodology was used to develop an embeddable version of a system called the Flight Control Maintenance Diagnostic System (FCMDS). It uses a model flight control system, a set of diagnostic rules, and a list of fault indications to generate a diagnosis and suggested corrective action. FCMDS was reimplemented in Ada to verify the methodology and demonstrate the feasibility of using Ada for embedded AI systems.

2 INTRODUCTION

2.1 The Need for AI In Embedded Systems

Modern military aircraft require increasingly complex and capable software. One way to meet this demand is to use Artificial Intelligence technology. For example, the fault isolation system described herein must do more than provide a list of instrument readings: it must also provide a diagnosis and suggested corrective action. The diagnosis is based on many instrument readings and uses the kind of rules that a human expert would use.

Since these AI systems interface with aircraft systems, they must work with the avionics computer hardware and software environment. They must be written in a programming language compatible with the rest of the avionics software.

Progress has been made in Artificial Intelligence using programming languages and software development environments specially suited for AI. Lisp and Prolog are two important AI languages. Avionics languages are very different from artificial intelligence languages; the language mandated by the Department of Defense for new mission critical systems is Ada. This paper describes a methodology that will enable us to develop software in an AI environment and field the software in military aircraft.

2.2 Languages For AI (Lisp, Prolog)

AI languages share certain common characteristics. All are high level. The number of statements required to perform a function are many fewer than for a conventional language. Statements are problem-oriented and express *what* is to be done. This differs from lower level procedural languages, like Ada and Pascal, that express how a problem is to be solved.

Two common AI languages are Prolog and Lisp. Prolog is based on the predicate calculus. Lisp is a procedural language; its facility for symbolic processing and function-oriented control structure make it higher-level than other procedural languages.

Both languages are well suited for rapid prototyping, an important issue for AI systems since they are used for state-of-the-art problems. Mechanizing human intelligence is difficult, and the solutions are not well known enough for implementation without prototyping.

However, AI languages are not convenient for some of the requirements of mission critical software. The design objectives of AI languages are ease of use and the ability to conveniently describe complicated algorithms, while the design objectives of languages for mission critical software are fast execution, error detection, modularity, and the ability to interface with complex target architectures and non-standard peripherals.

2.3 Languages For Embedded Systems (Ada)

According to the Reference 5 and 6 DOD directives, Ada is the required language for mission critical software. In addition to the directive, there are technical

[*] Ada is a Registered Trademark of the U.S. Government, Ada Joint Program Office.

reasons that led to the selection of Ada as the implementation language. It is the right level language: high level enough to express complicated problems, and sufficiently lo\. level for integration into a system of many interfacing programs and much non-standard hardware.

Ada is a conventional language, very similar to Pascal, but with features that support software engineering. One of the major objectives of software engineering is the management of complexity. The Ada packages support modularity and data abstraction which aid in breaking down a problem into manageable pieces. We plan to use an AI system that attempts to capture human expertise. Even a small, simplified abstraction of human expertise can be very complex; thus the programming language used must support this complexity.

Ada supports parallel processing; all the functions needed for parallelism can be specified in Ada. There is no need to go out to the operating system or to use assembly language for parallelism. Although the first application of the methodology did not use parallel processing, it is expected that future applications will do so because of the heavy processing demands of artificial intelligence systems.

3 THE METHODOLOGY

3.1 Proposed Methodology

A standard life cycle for mission critical software is in DOD-STD-2167 (Reference 7). The proposed methodology suggests enhancing this life cycle by first prototyping the system in an AI language, then reimplementing the system in Ada. The steps of the methodology are:

1. Problem requirements analysis
2. High level software design
3. Build a prototype in Lisp or some other appropriate AI language such as ART
4. Reimplement in Ada using a library of AI functions in Ada
5. Test the Ada version in an simulated avionics environment.

For a project done in accordance with DOD-STD-2167, the prototyping would be part of the demonstration and validation phase. Depending on the size and complexity of the job, the prototyping could also be in the requirements or design phases. The prototyping and reimplementation steps are needed to take advantage of AI technology. Prototyping is important because artificial intelligence applications are usually advanced systems that are not well enough understood to be fully designed on paper. Typical applications are understanding human speech, automated design, or modeling human expertise to diagnose equipment. Experimentation is needed to get a working design. The prototype provides an executable design that can be used for experimentation and user verification.

The reimplementation step would be tedious if it were done from scratch. However, a library of subroutines significantly reduces the work. Many of these routines are generic, enabling them to be used in many applications.

This methodology has the advantage of being the best of both worlds. Prototyping is done in an AI language and the final installation is in an avionics language. The disadvantage is that a reimplementation is required from the AI language to Ada. Other methods that avoid this disadvantage were explored and found to be less useful. Alternate approaches are discussed in Section 5.

The described methodology was used to develop an embeddable version of a system called the Flight Control Maintenance Diagnostic System (FCMDS) (*). A design and prototype written in Lisp was available. FCMDS was reimplemented in Ada.

3.2 Lessons Learned

Lisp is a very different language from Ada, but there were surprisingly few problems encountered in converting FCMDS from Lisp to Ada. Some things that were accomplished in one line of Lisp required two or three lines of Ada. Some things that were available as language features in Lisp required a subroutine package in Ada.

The design and application knowledge are as important as the prototype. The prototype alone was insufficient because it is difficult to distinguish between procedures essential for the problem and peculiarities of the prototype language. The reimplementation had to be readable, maintainable Ada in good Ada style which could only result from a knowledgeable translation of the Lisp code based on the design as well as the prototype.

The reimplementation from Lisp to Ada did not differ greatly from a conventional design procedure in which there is an implementation from a graphic form (like data flow diagrams) to Ada. Deriving Ada code from flow charts or data flow diagrams is not normally considered a translation, but it is nonetheless subject to the misunderstandings and distortions of a translation process. The

----

* FCMDS was designed and prototyped in Lisp by Dr. Barbara Gilmartin and Thomas Burzesi of Grumman Aircraft Systems Division.

Lisp design, while not as readable as an equivalent graphic design, has the advantage of being executable.

3.3 Lisp Features in Ada

3.3.1 Dynamic Data Structures

A central feature of Lisp is a dynamic list composed of elements which may consist of single symbols (called atoms) or other lists. This very flexible data structure can be any length since the number of elements in a list can change at run time. A list can also be any depth because a list can be composed of elements that are themselves lists.

There is no intrinsic Ada data structure closely resembling a Lisp list. However, a package can easily be written to add a dynamic data structure capability to Ada. The structure can be an exact copy of the Lisp list, but for FCMDS, it was more convenient to write set and stack generic packages. These are less general than the Lisp dynamic list, but fit the problem description. Like the dynamic list, the set and stack have no size limit other than available memory. The size can be varied at run time.

These Ada data structures were used to model the diagnostic system as a varying number of Fault Records which themselves are of varying length. The generic stack package was used to represent the varying length records.

The set and stack packages were implemented using access types called pointers in other languages. The same features could have been implemented using arrays, but this would have placed a size limit on the structures.

3.3.2 Procedure As A Parameter In A Call

The primary control structure of Lisp is the function call. When a function is given a list of parameters, it is assumed that the first parameter is itself a function. This is one reason that Lisp looks so strange to someone used to conventional procedural languages.

For example, a Lisp square root function could be passed a numeric parameter, as in:

```
(Sqrt 25)
```

It could also be passed a function with parameters, as in:

```
(Sqrt (Min 35 37 42))
```

The function "Min" for minimum with its parameters is evaluated. The value is passed to "Sqrt". Ada has no equivalent feature. A function can be passed as a parameter to a generic procedure, but this is done during compilation and not at run time. The feature was simulated in Ada by passing an enumeration parameter and having the receiving routine use a CASE statement to select a function call. This is one instance where Lisp is more convenient and elegant than Ada. Nonetheless Ada was found adequate.

The treatment by Lisp of the first element of a list as a function name provides a very useful way to mix data and program. In one section of the FCMDS Fault Record there is a predicate field that can be either true or false, depending on whether or not a particular fault has occurred. In some cases, the value depends on whether or not some logical combination of faults have occurred. Typical FCMDS fault predicates in Lisp are:

```
(TRIGGERED Power-Failure)

(OR (TRIGGERED Power-Failure) (TRIGGERED
Actuator-Failure))
```

The first function is true if the function TRIGGERED with the parameter Power-Failure is true. Lisp normally takes the first element of a List as a function name. The second predicate is true if either sublist returns true. "OR" is a function name. Only the built-in Lisp logical functions and the natural control structure of Lisp are needed to produce any logical fault combination.

In English the Lisp expressions would be:

If Power-Failure is TRIGGERED ....

If Power-Failure is TRIGGERED or Actuator-Failure is TRIGGERED ....

The same effect is achieved in Ada, but requires additional programming. The Fault data record contains a predicate string with faults or a group of faults followed by logical operators, i.e., in reverse Polish notation. Examples are:

```
Power-Failure

Power-Failure Actuator-Failure Or
```

An Ada procedure called "Is_Cause" processes the predicates. The second predicate is true if either a power failure or an actuator failure has occurred. The reverse Polish notation is hard to visualize, though easy to program. While some Lisp constructs have any easy and direct Ada equivalent, this was not one of them.

Spoken English uses an infix notation where the operators like "or" are in the middle. Lisp uses a prefix notation in which operators preceed operands. The Ada solution uses a postfix notation; operators follow. The complexity of a statement is the same for all forms, and readability depends on what the reader is used to.

3.3.3 Weak Typing

Ada is a strongly typed language, while Lisp is a weakly typed language. In Lisp, a parameter of a function can be a character, or a string, or a floating point number, or an integer. In Ada, the type of a parameter must be declared in advance. Ada does not permit types to be mixed. The only way to add an integer number to a floating point number is to explicitly convert one type to the other.

Whether or not strong typing is better than weak typing depends on the application. Weak typing is flexible and is good for rapid prototyping. Strong typing permits early detection of errors - important for mission critical software. A common error is a mismatch of parameters between a calling program and a subroutine. In a strongly typed language, this error will be caught at compile time. In a weakly typed language, the error will not be caught until the program is running. Run time errors are much more expensive, and no airplane ever crashed because of a software error caught during compilation.

The Lisp version of FCMDS had almost no data definitions, as opposed to the Ada version which had three packages of data definitions and other data definitions in procedures. The need for data definitions increased the size of Ada programs, but also increased the readablity. The nature of Lisp is that programs are almost data free. Information is passed as values returned by functions. Thus, reimplementing Lisp as Ada requires data design.

4 FLIGHT CONTROL MAINTENANCE DIAGNOSTIC SYSTEM

The Flight Control Maintenance Diagnostic System (FCMDS) was chosen as the first test case for the methodology because it is a well-bounded problem. The AI language prototype was available, and there is much interest in maintenance diagnostic systems.

The function of FCMDS is to diagnose a list of fault words. It has one permanent file, which is a description of the sub-system being monitored. FCMDS is designed to become a part of the flight control of the X-29 experimental aircraft. Fault words are logically equivalent to lights on a control panel. They are generated by the hardware when a fault occurs. There are usually many fault words in one diagnosis. A data flow diagram of FCMDS can be seen in Figure 1.

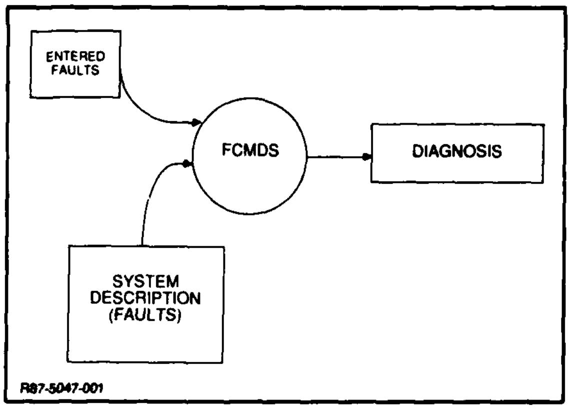


Figure 1 – Top Level Data Flow Diagram of FCMDS

Three fault records are shown in Figure 2, which is a simplified version of a system description. These constitute the fault model of a very simplified flight control system and represent possible faults in the system. A list of entered faults, in this case from one to three faults, is

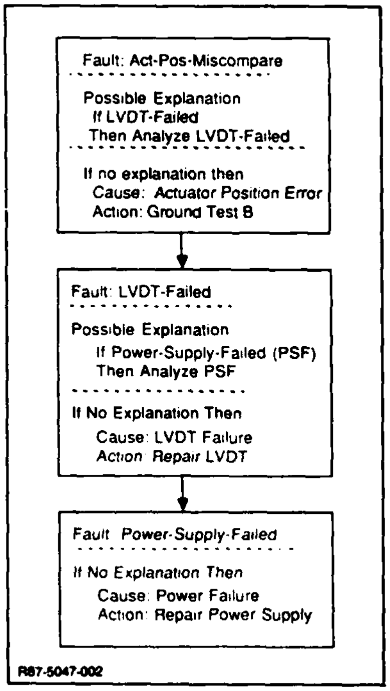


Figure 2 – FCMDS Simplified System Description

the variable input for a diagnosis. A fault records which matches an entered fault is said to be triggered.

All the triggered faults are analyzed in sequence. The triggered faults contain a list of possible explanations. They are in the form of:

IF predicate THEN action

The predicate is true if a designated fault or logical combination of faults is triggered. "Then" is either a cause-action combination or an instruction to analyze another fault.

If the predicate is true, the action taken depends on "Then". If it is a Cause-Action pair, then that pair becomes part of the diagnosis. The Cause indicates what caused the fault. The Action tells maintenance personnel what to do. If "Then" is a second fault, that fault is analyzed. The diagnosis of the second fault becomes the diagnosis of the first fault. The recursion of the solution comes from an analysis of a first fault requiring the analysis of a second fault.

The Ada version of FCMDS is shown in the data flow diagram of Figure 3. The first procedure "Initialize_Faults" (INIT_FLT) creates the knowledge base from text file data. It uses the Set package to build the set of faults. This set is a model of the equipment that will be diagnosed.

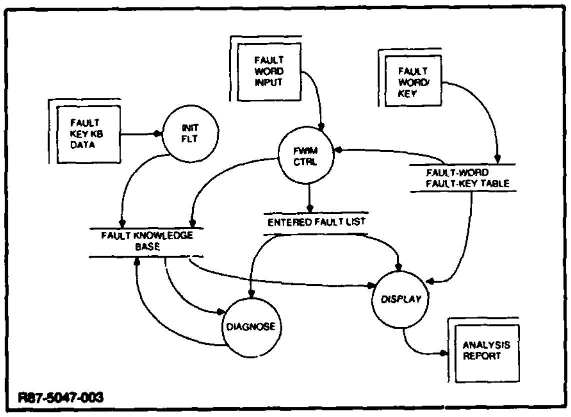


Figure 3 – FCMDS Ada Version Data Flow Diagram

"Fault_Word_Input_Model_Control" (FWIM-CTRL) takes the list of fault words and converts them to keys to the fault set. Fault words are coded indicators that can be likened to red lights on a maintenance panel. The keys to the fault set are meaningful names like "Power Failure." In addition, fault records for which there is a fault word are marked triggered.

"Diagnose" is the critical procedure. It examines all the triggered faults and derives a diagnosis consisting of a cause and an action. The Ada program is the same in principle as the Lisp prototype although the structure is different.

4.1 Size Comparison

The Lisp prototype was 644 lines of Lisp. The Ada equivalent was 2146 lines of Ada, a ratio of 3.3 to 1. The true ratio is actually higher because there was some input data verification in the Lisp version that was not carried over to the Ada version. Part of the reason for the difference is the extensive data definition required by Ada versus almost no data definition in Lisp.

The size difference does show that Lisp is a higher level language than Ada. This is not immediately obvious because Lisp is procedure, not problem oriented. It also shows that there is economy in prototyping in Lisp. If there were no expansion factor, it would not pay to prototype in Lisp and implement in Ada.

The size comparison between Ada and Lisp is based on a single data point and should be taken only as an indication.

5 ALTERNATE APPROACHES

A number of alternates to the proposed methodology were considered.

5.1 Automatic Translation Of Lisp To Ada

An automatic translation is very attractive because it is much quicker than a manual translation. However no satisfactory automatic translator was available and experience with other translators has shown that generated code is not readable and maintainable.

FCMDS is designed to be a small part of a larger software system. While FCMDS uses artificial intelligence techniques, the bulk of the system is conventional software. The artificial intelligence parts must be integrated and maintained along with the conventional software. In the environment for which FCMDS is intended, the conventional software is written in Ada.

The diagnostic software must be integrated, verified, and tuned. This requires a well-designed software product. No automated translation meets the requirements.

Lisp uses very little data compared to an Ada program doing the same task. An automatic translation system will do the job of producing Ada code from Lisp code, but it does not have enough knowledge to find meaningful data structures.

5.2 Embedded Lisp Machine

Another way to avoid prototyping in one language and implementing in another is to implement in the prototype language. Putting Lisp into an aircraft would eliminate the need to work in multiple languages for AI.

Lisp can run on conventional machines. There are Lisp systems that run on VAX computers and IBM PCs. However, Lisp is most efficient on machines that have been optimized for Lisp. Texas Instruments has a version of the Explorer system designed to be embedded into an avionics system.

There are several reasons why the Embedded Lisp Machine was not chosen as the recommended methodology. Most of the avionics environment is not AI. Introducing Lisp would introduce another language into the already complex avionics systems. Lisp is good for AI and rapid prototyping, but is not good for interfacing with buses, or for real time response. These are important for mission critical systems.

5.3 Prototype In Ada

Another way to eliminate the need to work in two languages is to prototype and implement in Ada. Ada alone is too low level to be considered a rapid prototyping language. However, with a library of subroutines and a good environment, the speed of generating a prototype in Ada can be increased.

Ada is not designed to be a prototyping language. The design goals of Ada are the opposite to those of a prototyping language. Some of the design goals and their consequences are:

* Readability (as opposed to writability)

* Maintainability (at expense of the initial build)

* Strong Typing (as opposed to free form data).

6 CONSIDERATIONS IN FIELDING AI-ADA SYSTEMS IN AVIONICS

The Maintenance Diagnostics System was not carried to the point where it was installed in a flying avionics system. It was converted to Ada and tested on a laboratory VAX computer. Some differences between a laboratory system and actual avionics is the critical response time, memory limits, and the need to never abort a program.

6.1 Using Recursion In Real Time Mission Critical Software

An important technique in AI programs, and Lisp programs in particular, is recursion. This is a powerful technique for controlling complexity. There is a constant tradeoff in software between many simple statements and a few complex statements. In many situations, the few complex statements that result in an overall reduction of complexity are recursive. FCMDS is naturally recursive. For example, a common situation would be one in which the analysis of the fault in the linear voltage difference transducer requires an analysis of the power supply. In software, this becomes the analysis routine with LVDT as a parameter calling itself recursively with the power supply as a parameter. This could be done without recursion, but would be much more complicated.

The negative side of recursion is that the amount of main memory used depends on the depth of recursion. If the analysis of the actuator recursively calls the analysis of the transducer, which recursively calls the analysis of the power supply, then three copies of the routine must be active at once. Since the depth of the recursion depends on the input data, the depth, and consequently the amount of storage used is variable and difficult to predict. At some depth, the system runs out of storage.

A mission critical system would have to be designed to never run out of storage, or to gracefully degrade when it reaches the storage limit. The Ada language, like Lisp, supports recursion. There is no basic difference in the way that each language handles the storage problem. The difference is between the laboratory environment and the mission critical environment. In the laboratory, storage is plentiful and an abrupt program crash does little harm, whereas in a mission critical environment, the exact opposite is true.

6.2 Using Access Types

FCMDS uses many dynamic data structures. The number of fault records is variable. The number of possible explanations in a fault record is variable. The number of terms in a predicate is variable. The variable size records are implemented by pointers. These are called access types in Ada. With these access types, memory is allocated as needed and released when it is no longer needed. A possible problem occurs if the memory that is released by the application programs is not reclaimed by the system: eventually the memory space is exhausted. This can be annoying or disastrous, depending on how critical the program is. The process of reclaiming memory is called garbage collections.

Lisp has one main data type–the dynamic list based on pointers. Garbage collection, therefore, has been well recognized as a requirement and is implemented in all known Lisp systems. In the Ada language definition of Reference 2, garbage collection is an implementers option. There is a facility called Unchecked-Deallocation that explicitly releases memory. This was used in the Ada version of FCMDS. Memory was released under program control.

Based on FCMDS experience, it would be extremely difficult to reimplement a Lisp system without using access types. Therefore an Ada implementation that is used for mission critical AI systems must have garbage collection.

6.3 Avionics Hardware

The Ada version of FCMDS was implemented on a DEC VAX 11/780 computer under VMS 4.2 using the DEC Ada compiler. The actual embedded version will likely be targetted to 1750A, MC68020 or i80386-based hardware. A MIL STD 1750A architecture machine is one possibility because it is an Air Force Standard. However, the 32-bit processors provide additional computer power needed for AI.

7 CONCLUSION

7.1 Evaluation Of The Methodology

This experiment has shown that Lisp systems can be reimplemented in Ada. It also showed that AI can be done in Ada. The methodology of prototyping in Lisp and reimplementing in Ada was a practical methodology for this test case.

8 ACKNOWLEDGMENTS

The maintenance and diagnostic system was designed and prototyped by Dr. Barbara Gilmartin and Thomas Burzesi. The authors would like to thank Martin Lewis and Charles Mooney for advice and guidance in the work described here.

9 REFERENCES

1. Winston, P.H. and Horn, B.K.P., "Lisp", Addison-Wesley, 1984.

2. "Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815 A, US Department of Defense, 1983.

3. Barnes, J.G.P., "Programming in Ada", Addison-Wesley, 1984.

4. Richardson, J.J., "Artificial Intelligence in Maintenance", Noyse Publications, 1985.

5. OPNAV Instruction 5200.28, Department of the Navy, Office of the Chief of Naval Operations, Washington, D.C. 20350.

6. AF Regulation 800-14, Department of the Air Force, Headquarters of the Air Force, Washington DC 20330, 29 September 1986.

7. DOD-STD-2167, "Defense System Software Development", 4 June 1985.

8. Dietz, D.C., "Ada Lisp: A Tool for Artificial Intelligence Implementation," IEEE, 1984.

9. Nadel, D, "Ada and Embedded AI", Defense Electronics, April 1986.

10. Sammet, J., "Why Ada is Not Just Another Programming Language", Communications of the ACM, August 1986.

11. Brauer, D.C., Roach, P.P., Frank, M.S., Knackstedt, R.P., "Ada and Knowledge Based Systems: A Prototype Combining the Best of Both Worlds", Expert Systems in Government Symposium, The Computer Society of the IEEE, October 22, 1986.

12. Clayton, B.D., "Art Programming Tutorial, Volume One: Elementary ART Programming", Inference Corporation, 1985.

13. "Texas Instruments Nears Completion of Lisp Language Microprocessors", Aviation Week and Space Technology", February 17, 1986.

10 LIST OF FIGURES

Teresa Doran
Grumman Aircraft Systems
Mail Stop C38-25
Bethpage, NY 11714
(516) 346-6446 or 575-2775

Teresa Doran is a Senior Software Engineer with the Grumman Aircraft Systems Division of Grumman Corporation. Ms. Doran has been involved with the Ada and Artificial Intelligence Technology Projects since 1986, and is currently working with Mr. Abrams to investigte Ada solutions for embedded AI software development. She has also been instrumental in establishing the Grumman Ada Users Group. Ms. Doran is a graduate of St. John's University.

Bernard Abrams
Grumman Aircraft Systems
Mail Stop C23-35
Bethpage, NY 11714
(516) 346-6445

Bernard Abrams is an Engineering Specialist at Grumman Aircraft Systems. He is currently working on the application of the Ada language to Artificial Intelligence problems. Mr. Abrams has served on KITIA (DoD-sponsored advisory committee on interfaces to Ada programming support environments), on which he has published a number of articles. Additionally, Mr. Abrams is an Adjunct Professor at the New York Institute of Technology where he teaches Software Engineering and System Analysis. He is a graduate of Rensselaer Polytechnic Institute and the Massachusetts Institute of Technology.

AUTHORS INDEX

END
DATE
FILMED
5-88
DTIC